DO MOBILE AGENTS NEED THE PHONE SCREEN?

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in mobile agents are dominated by the GUI paradigm, in which agents perceive UI information and emit screen interactions. However, mobile platforms also expose a command-line interface (CLI) with direct access to device state and services, so we study whether mobile agents can complete tasks without UI information. We evaluate three off-the-shelf coding agents across four model APIs on AndroidWorld and MobileWorld against two mobile-post-trained GUI specialists and two generalists, reproduced under a unified protocol, and construct hand-verified oracle solutions. We find that 88.8% of AndroidWorld and 86.3% of MobileWorld tasks are solvable without UI information, and every remaining task involves media or UI-only state, demonstrating that most benchmark tasks do not require the GUI. To separate interface from model, we fix the backbone at Opus 4.7: on CLI-solvable tasks, Claude Code matches its GUI counterpart (80.9% vs. 79.3% and 60.1% vs. 58.1%), so the GUI agent leads only on GUI-only tasks. Without mobile post-training, Claude Code beats every reproduced GUI specialist on both benchmarks (71.8% and 51.9%), and **MobileCLI**, our harness tuned on MobileWorld, lifts that backbone from 80.9% to 89.0% on held-out AndroidWorld's CLI-solvable tasks. Across an OS upgrade from Android 13 to 17, CLI agents change by at most 2.8 points, while the two GUI specialists drop 28.5 and 33.0 and the Opus 4.7 GUI agent only 4.0, suggesting that mobile post-training fits screens rather than tasks. For everyday intents the touchscreen was not designed to serve, we introduce the **Beyond-GUI Intent Suite**: 45 templates in five categories. Every CLI agent outperforms every GUI baseline in all five. The CLI is thus a first-class complement to the GUI, and the interface should be chosen per task. We release all agents, oracles, the suite, and infrastructure at `https://anonymous.4open.science/r/CLI_Agent-47D8`.

1 INTRODUCTION

Mobile agents are autonomous systems that interact with smartphones via natural language to complete user tasks (Wu et al., 2024; Hu et al., 2025; Liu et al., 2025). The dominant paradigm treats this as a vision problem: agents observe the phone screen and produce GUI actions, including tapping, swiping, and typing, to navigate applications, mimicking human behavior. Substantial research effort has advanced this paradigm across benchmarks, models, training algorithms, and infrastructure (Bai et al., 2024; Zhou et al., 2025; Wang et al., 2025a; Xu et al., 2025a; Gu et al., 2026; Xu et al., 2026; Team et al., 2026b).

However, the GUI paradigm is not the only way mobile agents can operate on mobile platforms. Android is a Linux-based platform that exposes a full command-line interface (CLI) through the Android Debug Bridge (ADB), allowing programmatic access to device services and data without ever rendering a pixel (Google, 2025a). The CLI offers three advantages. First, ADB can expose device states, including battery history, notification logs, and background processes, that no app screen surfaces (Google, 2025b), enabling the CLI paradigm to reach beyond what GUI agents can observe. Second, frontier coding agents (Anthropic, 2025; Singh et al., 2025; MiniMax, 2026) have demonstrated strong capabilities on challenging long-horizon terminal tasks in Linux environments (Merrill et al., 2026), operating in a text-only modality native to large language models. Third, generalization across OS and app versions remains a challenge for post-trained GUI agents (Yao et al., 2026; Liu et al., 2026), whereas a CLI agent never observes the screen and should be largely unaffected. Motivated by these observations, we argue that the CLI deserves first-class consideration

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

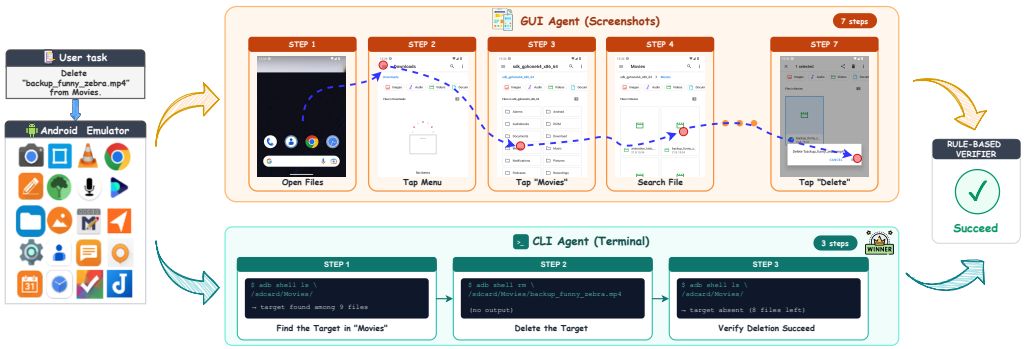


Figure 1: **GUI vs. CLI execution traces on a representative mobile task.** Both agents are asked to delete `backup_funny_zebra.mp4` from the Movies folder and proceed through three phases—*find the target*, *delete the target*, *verify deletion*. The GUI agent (top) consumes screenshots and emits taps, unfolding the phases over 7 UI steps (open Files → navigate menu → enter `Movies` → search → confirm delete). The CLI agent (bottom) consumes terminal `stdout` and emits ADB shell commands, collapsing the same phases into 3 steps (`ls`, `rm`, `ls`). A rule-based verifier scores both trajectories on their final state, holding the task and success criterion fixed.

as a complementary paradigm alongside the GUI for mobile agents, and we test the claim by answering two questions. **Q1 (viability):** Can mobile tasks be completed without UI information, competitively? **Q2 (complementarity):** Are there everyday user intents better served by the CLI but ignored by existing GUI-designed benchmarks?

To answer Q1, we evaluate three off-the-shelf coding agents (Claude Code (Anthropic, 2025), Terminus-2, and mini-swe-agent (Yang et al., 2024), all validated on Terminal-Bench (Merrill et al., 2026)) on AndroidWorld (Rawles et al., 2025) and MobileWorld (Kong et al., 2025) against four reproduced GUI baselines (GUI-Owl-1.5-32B (Xu et al., 2026), MAI-UI-8B (Zhou et al., 2025), Qwen3-VL-32B (Bai et al., 2025), and Claude Opus 4.7 driven end to end from screenshots), under the same protocol with identical tasks, rule-based verifiers, and evaluation metrics. Each CLI agent receives only terminal output, emits bash commands, and has no mobile-specific training. To adapt them to Android, we write a system prompt and general-purpose tools derived from public developer documentation, neither containing task- or app-specific information. To establish the CLI paradigm's ceiling, we construct hand-verified oracle solutions, reviewed by three external professionals, showing which tasks are CLI-solvable in AndroidWorld and MobileWorld. To test whether the gap to that ceiling is harness design, we also build **MobileCLI**, a purpose-built harness tuned on MobileWorld with Opus 4.7 and evaluated unchanged on both benchmarks, so AndroidWorld is held out. To evaluate generalization across OS and app versions, we re-run all agents on a sibling AndroidWorld environment that upgrades the OS from Android 13 to 17 and 17 of 20 apps, holding tasks, verifiers, and step budget fixed, and replay the oracle solutions there unchanged with no model in the loop.

To answer Q2, we evaluate both CLI and GUI agents on tasks that the GUI paradigm was not designed to accommodate. Since existing benchmarks were designed around the GUI paradigm and consequently exclude user intents the touchscreen serves poorly, we introduce the **Beyond-GUI Intent Suite**: 45 LLM-generated, human-verified task templates spanning five categories. Three categories have operations the app UI does not expose at scale: *bulk operations* ("delete every `.apk` file in Downloads"), *multi-condition filtering* ("list expenses above my average spending"), and *aggregation and top-K* ("top-3 expense categories this month"). The remaining two categories address state the app layer does not surface: *cross-app workflows* ("did anyone text me during my meetings yesterday?") and *hidden device state* ("which apps have been granted location permission"). Each task corresponds to an AndroidWorld-compatible rule-based verifier and a hand-verified oracle CLI solution for reliable assessment.

Our experiments support the CLI's first-class status as a complement to the GUI. For Q1, hand-verified oracle solutions make 88.8% of AndroidWorld (AW) and 86.3% of MobileWorld (MW) tasks solvable without UI information, so most benchmark tasks do not require the GUI. Claude Code with Opus 4.7 achieves 71.8% and 51.9%, above every reproduced GUI specialist but below

the same backbone driven end to end from screenshots (75.0% and 57.0%); all four off-the-shelf CLI configurations on AW land between 55.2% and 71.8%, so the result is not a single-system artifact. Holding the backbone fixed at Opus 4.7, the CLI agent matches its GUI counterpart on the CLI-solvable subset (80.9% vs. 79.3% and 60.1% vs. 58.1%), so the GUI agent's full-set lead comes from the GUI-only tasks. Harness engineering is a large lever. Tuned on MobileWorld and run unchanged on AndroidWorld, MobileCLI lifts the same backbone from 80.9% to 89.0% on AW's CLI-solvable tasks, closing about 40% of the distance to the oracle ceiling. Upgrading Android 13 to 17 changes CLI agents by at most 2.8 points while the two GUI specialists lose 28.5 and 33.0 and the frontier generalist only 4.0, suggesting that mobile post-training fits screens rather than tasks; replayed with no model in the loop, 93% of shell solutions still pass against 58% and 42% for UI actions. For Q2, every CLI agent outperforms every GUI baseline on the Beyond-GUI Intent Suite (60.7–74.1% vs. 22.2–41.5%) in all five categories. A trajectory-level error analysis shows the two paradigms fail in distinct ways.

2 Related Work

Mobile GUI agents. Mobile GUI agents aim to complete natural-language tasks on smartphones by perceiving screen state and emitting touch, swipe, and type actions, a setting that couples long-horizon decision making with fine-grained visual grounding under variable layouts and dynamic app state (Wu et al., 2024; Hu et al., 2025; Liu et al., 2025). Prior work falls into two categories. Agentic frameworks prompt foundation models, typically proprietary, within pipelines that plan, act, reflect, and remember over screenshots or UI trees (Zhang et al., 2025b; Wang et al., 2024b;a; Ye et al., 2025; Wen et al., 2024; Guo et al., 2026). End-to-end approaches instead post-train open models, first with supervised or preference fine-tuning on GUI grounding and trajectory data (Hong et al., 2024; Cheng et al., 2024; You et al., 2024; Wu et al., 2025; Xu et al., 2024; Lin et al., 2025; Qin et al., 2025; Yang et al., 2025; Dai et al., 2026) and, more recently, with reinforcement learning, often online in live Android environments (Bai et al., 2024; Shi et al., 2025; Gu et al., 2025; Wang et al., 2025a; Xu et al., 2025a; Zhou et al., 2025; Yan et al., 2025; Team et al., 2026b; Xu et al., 2026; Gu et al., 2026). However, both categories mainly share the same I/O interface: the observation is the rendered UI (screenshot pixels or accessibility tree) and actions are grounded in rendered UI elements, which bounds performance by grounding accuracy, limits observation to state that some screen renders, leaves agents brittle to layout shifts (Yao et al., 2026; Liu et al., 2026), and forces bulk or cross-app operations into long tap sequences.

Mobile agent benchmarks. Mobile agent benchmarks fall into two categories. Offline datasets compile human-annotated Android trajectories and grade agents by per-step action matching against the human references (Rawles et al., 2023; Li et al., 2024), but static demonstrations restrict evaluation to step-level accuracy and cannot capture trajectory-level task success. To address this, online interactive benchmarks boot a live Android emulator and grade trajectory-level success through rule-based or LLM-as-judge verifiers (Rawles et al., 2025; Xu et al., 2025b; Wang et al., 2024c; Chen et al., 2024; Chai et al., 2025); MobileWorld (Kong et al., 2025) further extends this setting with broader app coverage, networked client-server applications, longer-horizon and cross-application tasks, agent-user interaction, and MCP-augmented environments. However, tasks in these benchmarks largely operationalize the user intents the screen can surface, introducing a sampling bias that leaves a broad class of real user needs unmeasured; our Beyond-GUI Intent Suite (§4) targets this gap. AppWorld (Trivedi et al., 2024) similarly evaluates daily multi-app tasks, but through 457 simulated APIs rather than Android's terminal, and without comparison to GUI agents.

Coding agents and programmatic control. Coding agents issue text-based actions through programmatic interfaces such as shell commands and tool calls, evaluated on software-engineering and OS tasks (Wang et al., 2024d; Yang et al., 2024; Wang et al., 2025b); frontier coding agents (Anthropic, 2025; Singh et al., 2025; MiniMax, 2026) achieve strong performance on these benchmarks through text-only interaction. Harness engineering — the design of tools, system prompts, context management, and feedback loops around an off-the-shelf agent — has emerged as the principal mechanism for adapting such agents to new domains without retraining (Lopopolo, 2026; Lee et al., 2026b). Terminal-Bench (Merrill et al., 2026) demonstrates that coding agents can solve realistic system and developer tasks on Linux through the shell alone. Closest to our work, hybrid systems extend the GUI action space with shell or code commands to improve capability (Du et al., 2026; Li

et al., 2026; Lee et al., 2026a); UI information remains available at every step, and none systematically studies a condition in which the agent acts without it. Comparisons of programmatic and GUI control have been conceptual (Zhang et al., 2025a) or confined to the web and desktop (Song et al., 2025; 2026; Zhou et al., 2026b), where they reach conflicting conclusions; mobile platforms remain unexamined. We provide this comparison on Android, treating the CLI as a standalone paradigm alongside the GUI.

3 Experimental Setup

We define an **agent configuration** in each paradigm by three parts: an **I/O interface**, the screen in the GUI paradigm and the shell in the CLI paradigm; a **model backbone**, either a frontier model API or a mobile post-trained specialist; and a **harness** that couples backbone to interface through tools, a system prompt, context management, and a feedback loop.

Benchmarks and Baselines. We evaluate on two interactive mobile-agent benchmarks whose rule-based verifiers check device state directly, ensuring compatibility with terminal-only execution. AndroidWorld (Rawles et al., 2025) provides 116 task templates across 20 Android applications, parameterized by random seeds. MobileWorld (Kong et al., 2025) extends this setting with broader app coverage spanning networked client-server applications in e-commerce, enterprise communication, and social media, and with longer-horizon, cross-application tasks (27.8 vs. 14.3 average steps per task; 62.2% vs. 9.5% multi-app tasks). We evaluate on 117 of its tasks, excluding the user-interaction and MCP (Hou et al., 2026) subsets that fall outside the scope of a GUI-vs-CLI comparison. We exclude AndroidLab (Xu et al., 2025b) because its XML-based verifier requiring UI state is incompatible with the CLI paradigm.

Following CoAct-1 (Song et al., 2026; Zhao et al., 2026), we report success rate (SR) and average steps per task across three random seeds, with mean and standard deviation reported. For GUI baselines, we reproduce four agents that emit screen actions directly: two open-weight specialists post-trained on mobile datasets (GUI-Owl-1.5-32B (Xu et al., 2026) and MAI-UI-8B (Zhou et al., 2025)), one open-weight generalist (Qwen3-VL-32B (Bai et al., 2025)), and one frontier generalist (Claude Opus 4.7). In addition to these *reproduced* baselines, we report leaderboard results for UI-Venus-1.5 (Team et al., 2026b), MobileRL-9B (Xu et al., 2025a), Qwen-UI-Agent (Zhou et al., 2026a), and Kimi K3 (Team et al., 2026a). None of these systems discloses its evaluation pipeline, so we cite them as reference points rather than re-running them under our protocol.

CLI Agents. We use three off-the-shelf coding agents spanning two design categories: a frontier agent (Claude Code (Anthropic, 2025)) representing state-of-the-art coding capability, and two minimal-scaffold agents (Terminus-2 (Merrill et al., 2026) and mini-swe-agent (Yang et al., 2024)) designed for Terminal-Bench command-line tasks with simple harness architectures, chosen to evaluate how effectively such designs transfer to mobile platforms. Specifically, each agent operates in a text-only Android terminal: it receives the task instruction and terminal output at each step (e.g., system state, database query responses, or error messages) as input, and emits ADB commands to interact with the device. No visual or structural UI information such as screenshots or accessibility trees is used at any step. Appendix A.3 compares what each paradigm observes, its privilege level, and whether it is available on a consumer device. Figure 1 contrasts CLI and GUI execution on a representative task.

System Prompt and Tools. Whereas GUI agents adapt to Android via post-training on large mobile GUI datasets, no comparable public corpus exists for Android-CLI interaction. We therefore adapt general coding agents through a system prompt derived from public Android developer documentation and engineering practices, which prescribes a discover–inspect–act–verify interaction cycle, a prioritized hierarchy of mechanisms for modifying device state, efficiency strategies, and platform-specific persistence patterns, but contains no task-specific or app-specific information that could leak to CLI agents (full prompts in Appendix A.1). Motivated by recent advances in harness engineering (Tian et al., 2026; Lopopolo, 2026; Lee et al., 2026b), we additionally provide general-purpose tools that wrap raw ADB shell commands into higher-level operations: four on AndroidWorld (SQLite queries, file read and write, and file search) and twelve on MobileWorld, whose backend services, structured data, and networked APIs call for more; four of the added tools act on MobileWorld's

backend servers rather than the device, and Appendix C.1 reports results separately for tasks that need them. Appendix A.2 provides complete tool specifications. Table 1 reports each harness in its best configuration, with the wrappers enabled; every analysis after it holds the `bash`-only configuration fixed, so that tool design does not vary across the comparisons we draw. Appendix C.3 reports the two configurations side by side.

Purpose-built harness. To test whether the gap to the CLI ceiling is harness design rather than a limit of the interface, we construct **MobileCLI**, a `bash`-only harness that incorporates device-state context, mobile-specific error feedback, and a budget-aware control loop (Appendix A.4). It is tuned on MobileWorld with Claude Opus 4.7 and contains no task-specific or app-specific information. We evaluate it unchanged on both benchmarks, so AndroidWorld is held out and the MobileWorld numbers are in-domain. Its configuration differs across the two benchmarks only in the action space.

Oracle Construction. To establish the empirical CLI ceiling on each benchmark, we construct oracle solutions through human–LLM collaboration. Each solution must satisfy two criteria: *solvability*, meaning the command sequence passes the rule-based verifier; and *integrity*, meaning the solution arrives at the answer through principled exploration rather than extracting it from verifier's source code. Each task is attempted up to five times with verifier feedback; every solution is independently reviewed by all three external professionals. This process yields 88.8% (103/116) CLI-solvable tasks in AndroidWorld and 86.3% (101/117) in MobileWorld. The remaining 13 tasks on AndroidWorld and 16 on MobileWorld involve interactions that are not CLI-solvable: interacting with visual content (e.g., transcribing receipts from photos, free-hand drawing) or capturing multimodal data (e.g., camera photos or audio recording); on MobileWorld this set additionally contains a handful of in-app panel-only fields (Google Maps detail labels, shopping-cart totals) that no public API exposes. Details of the construction protocol and leakage safeguards are provided in Appendix A.5.

4 BEYOND-GUI INTENT SUITE

Real mobile users have everyday needs the touchscreen interface was not designed for: cleaning up leftover files, asking which expense category dominated this month, or finding out which apps have been granted background-location access. Existing mobile-agent benchmarks were designed around the GUI paradigm, so user intents the touchscreen serves poorly fall outside their scope. This sampling gap leaves a measurable class of user intent invisible to benchmark scores. We introduce the Beyond-GUI Intent Suite, a set of **45 task templates across five categories** that target this gap directly, each with an AndroidWorld-compatible rule-based verifier and a hand-verified oracle CLI solution.

Taxonomy. The 45 task templates fall into five categories of mobile-user intent that current touchscreen UIs serve poorly. Three require operations the app UI does not expose at scale: **Bulk operations** (10 tasks) apply one change to many items at once (e.g., *"delete every .apk file in Downloads"*); **Multi-condition filtering** (10) selects items satisfying a conjunction of constraints app filter UIs rarely compose (e.g., *"list expenses above my average spending"*); and **Aggregation and top-K** (10) computes summaries over many rows no single screen displays (e.g., *"top-3 expense categories by total this month"*). Two require state the app layer does not surface: **Cross-app workflows** (9) combine state from multiple apps (e.g., *"did anyone text me during my meetings yesterday?"*), and **Hidden device state** (6) queries system-level facts no app exposes (e.g., *"which apps have been granted location permission"*).

Construction and Verification. The 45 task templates are LLM-generated and human-verified by three mobile users. Following AndroidWorld's automatic task-parameterization mechanism, each template yields a distinct task instance per seed. With three seeds $\{7, 30, 1234\}$, we construct 135 task instances. Success is checked by a binary rule-based script that reads the final device state, matches the agent's reported answer against ground truth, or both—with no LLM-judge in the loop. We verify each task along three axes. *Solvability*: applying the Section 3 oracle procedure, we construct a hand-verified CLI command sequence per task; all 45 pass the rule-based verifier. *Realism*: three annotators individually rate each task template on a three-level rubric (highly realistic / stretched / contrived); inter-annotator Cohen's $\kappa = 0.91$. *Additivity*: for each task, we compute the cosine similarity to its nearest AndroidWorld neighbour using `text-embedding-3-large`; the

Table 1: **Main results on AndroidWorld (AW) and MobileWorld (MW). SR** is success rate over the full task set and **SR-CLI** is success rate over the CLI-solvable subset (103/116 on AW, 101/117 on MW); **Steps** is mean agent steps per task. Cells are mean (std) across 3 seeds for every system we ran, and the best per column is in bold. *Reproduced* means we ran the system end to end under our own protocol. All off-the-shelf CLI agents run `bash` plus tools that wrap raw `adb shell` commands (Appendix A.2); MobileCLI is `bash`-only. **Oracle** is the CLI ceiling from hand-verified reference solutions. **MobileCLI** is tuned on MobileWorld with Opus 4.7, so AndroidWorld is held out and its MobileWorld numbers are in-domain. ⋆ Leaderboard-reported: SR is taken as published, since the evaluation code is unavailable and we cannot re-run these systems under our protocol. ‡ SR-CLI and Steps are computed from the authors' released trajectories. The full leaderboard is in Table 11.

		AndroidWorld			MobileWorld		
Method	**Model API**	**SR↑**	**SR-CLI↑**	**Steps↓**	**SR↑**	**SR-CLI↑**	**Steps↓**
GUI Agents (Leaderboard-reported)⋆							
Qwen-UI-Agent‡	-	-	-	-	**82.1**	82.2	25.1
MobileRL-9B	-	**80.2**	-	-	-	-	-
UI-Venus-1.5-30B-A3B	-	77.6	-	-	17.1	-	-
Kimi K3‡	-	-	-	-	74.4	74.3	25.4
GUI Agents (Reproduced)							
Claude e2e	Opus 4.7	75.0 (0.9)	79.3 (1.1)	21.4 (0.0)	57.0 (1.3)	58.1 (1.5)	28.6 (0.6)
GUI-Owl-1.5-32B	-	69.3 (0.5)	71.6 (1.5)	15.5 (2.5)	41.3 (4.4)	38.0 (4.5)	34.7 (1.0)
MAI-UI-8B	-	68.1 (0.6)	69.3 (0.3)	18.4 (1.2)	27.4 (4.5)	28.7 (6.0)	38.5 (2.0)
Qwen3-VL-32B	-	57.2 (3.0)	62.8 (3.9)	18.4 (0.7)	15.1 (2.0)	13.2 (1.1)	24.4 (0.1)
CLI Agents (off-the-shelf coding harnesses)							
Claude Code	Opus 4.7	71.8 (1.8)	80.9 (2.0)	15.2 (0.6)	51.9 (1.0)	60.1 (1.1)	30.2 (0.6)
mini-swe-agent	Sonnet 4.6	70.7 (2.3)	79.6 (2.6)	12.1 (0.3)	43.6 (2.6)	50.5 (3.0)	33.5 (0.4)
Terminus-2	GPT-5.3 Codex	63.2 (0.5)	71.2 (0.6)	**7.1 (0.1)**	36.2 (1.0)	41.9 (1.1)	24.5 (1.9)
Terminus-2	MiniMax M2.7	55.2 (3.8)	62.1 (4.2)	14.5 (0.5)	16.8 (1.3)	19.5 (1.5)	32.9 (0.8)
CLI Agent (purpose-built harness)							
MobileCLI	Opus 4.7	79.0 (0.5)	**89.0 (0.6)**	7.4 (0.3)	71.5 (1.0)	**82.8 (1.1)**	**18.3 (0.3)**
Oracle (CLI ceiling)	-	88.8	100.0	3.7	86.3	100.0	5.6

resulting distribution sits to the left of AndroidWorld's within-suite reference, with 0% exceeding similarity 0.7. Details are provided in Appendix B.

5 Results

5.1 Benchmark Performance

CLI agents are competitive with leading GUI agents, and the remaining gap is harness design. We first ask whether mobile tasks can be completed with no UI information at all, and completed competitively. Claude Code with Opus 4.7 reaches 71.8% on AndroidWorld and 51.9% on MobileWorld (Table 1), above every open-weight GUI baseline we reproduced, though below the same backbone driven end to end from screenshots (75.0% and 57.0%). The highest leaderboard-reported results, MobileRL-9B at 80.2% on AndroidWorld and Qwen-UI-Agent at 82.1% on MobileWorld; neither system discloses its evaluation pipeline, so we cannot verify these numbers under our protocol. The oracle ceiling is 88.8% and 86.3%, leaving the off-the-shelf harnesses 17 and 34 points short. To test whether that gap is harness design rather than a limit of the interface, we build MobileCLI on the same backbone: it reaches 79.0% and 71.5%, 7.2 and 19.6 points above Claude Code, and 89.0% and 82.8% on the CLI-solvable subsets in 7.4 and 18.3 steps. It is tuned on MobileWorld with Opus 4.7, so the AndroidWorld gain is held out while the MobileWorld gain is in-domain; on MobileWorld its subset score matches Qwen-UI-Agent's 82.2% while its full-set score trails 82.1%.

Competitiveness is robust across agent configurations, not an artifact of one system. To show that this is not an artifact of a single agent system, we evaluate off-the-shelf CLI configurations

Table 2: **Cross-version generalization from Android 13 to Android 17.** The Android OS and 17 of 20 apps are upgraded, while tasks, verifiers, and the 50-step budget are fixed. All rows use the bash-only configuration and report success rate (%) over the 116 AndroidWorld tasks: mean (std) over 3 seeds for agents, and the oracle replay runs the Android 13 oracle solutions once with no model. Δ is the change in success rate, in points.

Agent	I/O interface	Android 13	Android 17	Δ
Agent evaluation				
MobileCLI + Opus 4.7	CLI	79.0 (0.5)	80.5 (0.5)	$\uparrow$1.5
Claude Code + Opus 4.7	CLI	71.8 (0.5)	71.0 (4.1)	$\downarrow$0.8
mini-swe-agent + MiniMax M2.7	CLI	57.2 (3.3)	59.2 (1.0)	$\uparrow$2.0
Terminus-2 + GPT-5.3 Codex	CLI	48.9 (3.0)	51.7 (0.9)	$\uparrow$2.8
Claude e2e + Opus 4.7	GUI (screenshot)	75.0 (0.9)	71.0 (1.0)	$\downarrow$4.0
GUI-Owl-1.5-32B	GUI (screenshot)	69.3 (0.5)	40.8 (1.0)	$\downarrow$28.5
MAI-UI-8B	GUI (screenshot)	68.1 (0.6)	35.1 (1.3)	$\downarrow$33.0
Qwen3-VL-32B	GUI (screenshot)	57.2 (3.0)	50.6 (1.8)	$\downarrow$6.6
Oracle replay without a model				
CLI oracle	CLI	88.8	82.8	$\downarrow$6.0
GUI oracle	GUI (a11y tree)	100.0	57.8	$\downarrow$42.2
GUI oracle	GUI (screenshot)	100.0	42.2	$\downarrow$57.8

spanning three harnesses and four model APIs. On AndroidWorld their success rates all land between 55.2 and 71.8%, with mini-swe-agent and Sonnet 4.6 reaching 70.7% and matching GUI-Owl-1.5-32B (69.3%) in fewer steps (12.1 vs. 15.5). Tool wrappers change this picture only for GPT-5.3 Codex, the weakest backbone without them ($+14.3$ and $+11.8$ points, against at most 1.4 for every other pairing); the complete harness $\times$ model-API matrix is in Appendix C.3. On MobileWorld, the picture is more model-dependent: Claude Code with Opus 4.7 and mini-swe-agent with Sonnet 4.6 clear every reproduced GUI specialist, while the two Terminus-2 configurations (with Codex and with MiniMax) trail GUI-Owl, with the gap widening sharply for weaker model APIs on MobileWorld's more challenging, longer-horizon, cross-application tasks. Competitiveness is therefore a property of the paradigm as instantiated with coding-capable backbones, rather than of a particular harness or model API.

The same model with coding capability performs better through the CLI. To disentangle the model from the I/O interface, we compare the two paradigms at fixed model backbones under identical protocols (Table 12, Appendix C.2). With Opus 4.7, the off-the-shelf CLI agent trails the screen on the full set and matches it on the CLI-solvable subset ($+1.6$ points, within seed noise), while MobileCLI leads on both, by 4.0 and 9.7. With Qwen3-VL-32B the direction reverses: the same purpose-built harness reaches 34.8% against the screen's 57.2%, because that backbone has no coding-focused post-training. Thus the shell wins on CLI-solvable tasks only for a coding-capable backbone, and the backbone matters as much as the interface.

5.2 Generalization across OS and App Versions

We build a sibling AndroidWorld environment that upgrades the OS from Android 13 to 17 and 17 of 20 apps, holding all else fixed (Table 2). All four CLI agents, which never observe the UI, change by at most 2.8 points, whereas the two GUI specialists post-trained on mobile data lose 28.5 and 33.0; the open-weight generalist loses 6.6 and the frontier generalist only 4.0. Mobile post-training appears to fit screens rather than tasks. Fixing the backbone at Opus 4.7 separates model from interface: Claude Code loses 0.8 points against Claude e2e's 4.0. Replaying the Android 13 oracle solutions on Android 17 with no model in the loop, the CLI loses only 6.0 points against 42.2 and 57.8 for the accessibility tree and screen coordinates. CLI failures stem from app-private database state rather than public interfaces, which remain backward compatible; GUI failures stem from redesigned screens that no longer expose their contents to the accessibility tree, or whose shifted layouts send a fixed tap to a neighbouring element. Appendix C.4 provides further details.

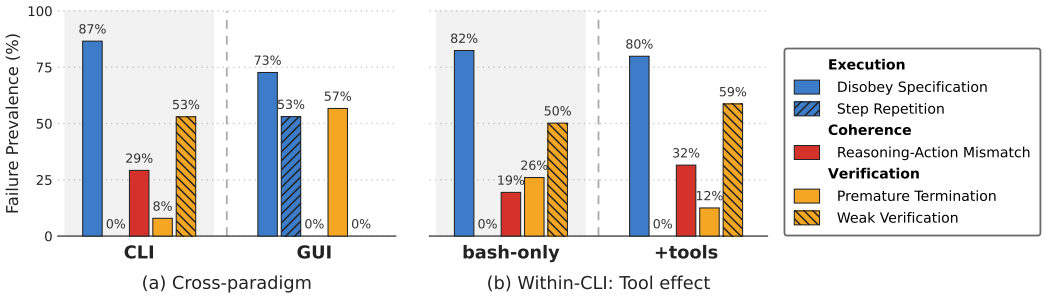


Figure 2: **Trajectory-level failure modes.** Each bar denotes the fraction of failed trajectories exhibiting the corresponding taxonomy category (a trajectory may exhibit multiple). **(a) Cross-paradigm:** CLI vs. GUI on the same 103 CLI-solvable tasks of AndroidWorld. **(b) Within-CLI tool effect:** GPT-5.3 Codex is evaluated under `bash-only` vs. `+tools` (four generic, structured tool calls). Color encodes the top-level category (Execution / Coherence / Verification); hatching marks Step Repetition and Weak Verification within their categories.

5.3 Trajectory-Level Error Analysis

Since average success rates do not reveal failure patterns, we conduct a trajectory-level error analysis along two comparison axes: (i) the GUI and CLI paradigms, over all four GUI baselines and the off-the-shelf CLI agents; and (ii) `bash-only` vs. `+tools` for GPT-5.3 Codex, selected for its largest tool-boosted SR gain in Appendix C.3. We adapt the Multi-Agent System Taxonomy (MAST) (Cemri et al., 2026) to our single-agent mobile setting by dropping inter-agent coordination subcategories (following Merrill et al., 2026) and mapping the remaining nodes onto mobile-platform failure modes, yielding a two-level paradigm-agnostic taxonomy: three top-level error classes (*Execution*, *Coherence*, *Verification*), each containing several named failure modes. (full taxonomy and per-mode examples in Appendix D.1). To identify candidate failure-mode clusters, we use Docent (Meng et al., 2025) to summarize each failed trajectory within each paradigm separately, yielding paradigm-specific clusters. To map clusters onto the named modes in the taxonomy, we apply Claude Opus 4.7 with a rubric calibrated against two human annotators. (full protocol and validation in Appendix D.2).

Figure 2(a) summarizes failure modes across the two paradigms, reporting percentages over failed trajectories. Both paradigms share *Disobey Specification* as the dominant failure mode (87% CLI / 73% GUI), but their tails diverge: GUI is dominated by *Step Repetition* and *Premature Termination*, CLI by *Reasoning-Action Mismatch* and *Weak Verification*. These distinct failure signatures suggest that the two paradigms require distinct strategies for improvement. Within CLI, the four tools cut *Premature Termination* from 26% to 12% on GPT-5.3 Codex while raising *Reasoning-Action Mismatch* (+13 points) and *Weak Verification* (+9 points): the tools prevent the agent from quitting early, so failed trajectories now exhibit downstream reasoning and verification errors (Figure 2(b)).

5.4 Beyond-GUI Intent Suite

We evaluate four CLI agents (bash-only) and four GUI agents on the 45-task Beyond-GUI Intent Suite (Section 4), with a maximum of 50 steps per task.

All CLI agents lead all GUI agents. As reported in Table 3, every CLI agent leads every GUI agent in mean success rate, by 19.2 to 51.9 points. The ordering holds in all five categories, though narrowly in aggregation/top-K (80.0 vs. 73.3) and cross-app workflows (25.9 vs. 22.2), where the weakest CLI agent and the strongest GUI agent sit within seed noise. Even the strongest CLI agent, MobileCLI with Opus 4.7, reaches only 74.1%, leaving substantial headroom below the 100% oracle ceiling.

Why do GUI agents fail? The cross-app workflow category exposes a structural ceiling for the GUI paradigm. Every GUI agent we evaluate caps at no more than 22.2%, suggesting that the bottleneck is maintaining state across multiple apps when each step observes only a single screenshot. Hidden device state exposes a second ceiling: every GUI agent averages 16.7%, because this state lives on

Table 3: **Per-category and overall performance on the 45-task Beyond-GUI Intent Suite (3 seeds; mean (std); SR in percent)**. Best per column in bold; ties in Bulk and Filter are bolded jointly. CLI oracle solutions reach 100% SR.

Agent	Bulk	Filter	Aggregation	Cross-app	Hidden	Avg SR	Steps
GUI agents							
Qwen3-VL-32B	33.3 (5.8)	20.0 (10.0)	36.7 (5.8)	0.0 (0.0)	16.7 (0.0)	22.2 (4.4)	12.9 (1.1)
MAI-UI-8B	56.7 (5.8)	36.7 (15.3)	10.0 (10.0)	11.1 (0.0)	16.7 (0.0)	27.4 (6.4)	23.4 (1.6)
GUI-Owl-1.5-32B	46.7 (5.8)	40.0 (10.0)	43.3 (5.8)	11.1 (0.0)	16.7 (16.7)	33.3 (3.8)	19.4 (1.0)
Claude e2e + Opus 4.7	50.0 (10.0)	33.3 (5.8)	73.3 (11.5)	22.2 (0.0)	16.7 (0.0)	41.5 (2.6)	19.2 (0.8)
CLI agents							
Terminus-2 + Codex	**76.7 (5.8)**	56.7 (5.8)	83.3 (5.8)	25.9 (12.8)	**55.6 (9.6)**	60.7 (6.4)	**5.7 (0.4)**
mini-swe-agent + MiniMax	73.3 (11.5)	76.7 (5.8)	80.0 (0.0)	48.1 (12.8)	44.4 (9.6)	66.7 (4.4)	15.4 (1.2)
Claude Code + Opus 4.7	**76.7 (5.8)**	**86.7 (5.8)**	83.3 (5.8)	40.7 (6.4)	44.4 (9.6)	68.9 (0.0)	11.0 (1.4)
MobileCLI + Opus 4.7	70.0 (0.0)	**86.7 (5.8)**	**90.0 (0.0)**	**63.0 (25.7)**	50.0 (0.0)	**74.1 (4.6)**	7.4 (0.1)

system screens these agents never navigate to. The two largest gaps appear in aggregation/top-K and multi-condition filtering, where mean CLI success exceeds mean GUI success by 43.3 and 44.2 points. Both require set-level operations such as composing predicates and summing across rows, which the CLI expresses as one shell pipeline but the screen only through long tap sequences. The CLI advantage therefore grows when tasks demand composition over many items rather than repetition of a single operation, where the gap narrows to 27.5 points.

6 Conclusion

We studied whether mobile tasks can be solved without UI information, treating the CLI as a first-class complement to the GUI. Under one protocol on AndroidWorld and MobileWorld, off-the-shelf coding agents are competitive with reproduced GUI baselines without mobile post-training, and on AndroidWorld remain more robust than mobile-post-trained GUI specialists across OS and app upgrades, suggesting that such post-training fits screens rather than tasks. Hand-verified oracle solutions demonstrate that most GUI benchmark tasks can be solved using CLI only. To separate interface from model, we fix the backbone at Opus 4.7: on CLI-solvable tasks the CLI matches the screen, so the GUI agent's lead comes from GUI-only tasks. Claude Code outperforms every reproduced GUI specialist on both benchmarks, and **MobileCLI**, our harness tuned on MobileWorld, lifts that backbone from 80.9% to 89.0% on held-out AndroidWorld's CLI-solvable tasks. The paradigms fail in distinct ways, motivating different remedies. On the Beyond-GUI Intent Suite, every CLI agent outperforms every GUI baseline in all five categories. The gap is widest where a task composes over many items rather than repeating one, and least sensitive to a stronger GUI backbone where the state it needs sits off the app's own screens.

Limitations. The CLI paradigm carries privacy and security risks: ADB grants shell-level access to app databases, files, content providers, and system state without Android's permission checks, cloud-hosted model APIs receive this data off-device at every step, and the privileged action surface admits destructive operations (Appendix E). Deployability is also constrained, since ADB requires developer mode or host-side provisioning and every configuration we report, including MobileCLI, runs against a hosted model API that costs more than everyday use can afford (Appendix A.3); our study is correspondingly Android-specific. Finally, a CLI agent operates only on state reachable from a terminal, so functionality hidden behind proprietary UI flows and inherently visual workloads such as video and image editing are not CLI-solvable.

Future Work. These gaps motivate two complementary directions. First, a small coding-agent model post-trained for the Android terminal would carry platform knowledge in its parameters rather than in the context window at every step, reducing inference cost and allowing on-device deployment so that user-private data never leaves the phone. Second, hybrid agents that add shell commands to a GUI action space are beginning to appear, but when a task, or a step within a task, should be routed to the CLI rather than the screen remains an open question for mobile agents.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks that require disclosure: implementing methods (coding assistance under our instructions); generating synthetic data (the Beyond-GUI Intent Suite task templates, and candidate oracle trajectories proposed by a Claude Code oracle-agent and validated by human reviewers; Appendix A.5); reformatting data (replacing GUI screenshots with one-sentence Claude Opus 4.7 captions before clustering; Appendix D.2); and supporting qualitative and thematic data analysis (Claude Opus 4.7 per-trajectory failure summaries, Docent clustering, and Claude Opus 4.7 mapping of clusters to taxonomy leaves under a rubric calibrated and validated against human annotators; Section 5.3, Appendix D.2). We have not used generative AI tools to develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, or interpret results; formulating mathematical claims, providing ingredients for or writing proofs, and translation are not applicable to this work. Additionally, we used generative AI tools as a writing assistant to improve readability. We have reviewed all AI-assisted work. All AI-assisted code was reviewed by us and verified through testing and by checking that experimental outputs behaved as expected; all reported results come from experiments we ran and analyzed ourselves. The generated BeyondGUI suite was manually inspected by us for correctness and consistency with our specifications. AI-suggested wording edits were limited to grammar, phrasing, and clarity, and we reviewed each edit to ensure it preserved the intended technical meaning. All research ideas, experimental designs, analyses, and claims are our own. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies mobile agents that operate Android devices through the shell. All experiments run on emulators and containerized backend services populated with synthetic benchmark data; no real user data was read, transmitted, or stored. The CLI paradigm itself carries deployment risks that we document rather than resolve (Section 6, Appendix E). Shell-level access bypasses Android's per-resource permission checks, cloud-hosted models receive whatever the agent reads, and the action surface includes irreversible operations such as `pm clear`, which our error analysis observed agents issuing as a blanket recovery step. Our harnesses block `adb pull` and `push`; the emulator shell is already root, as is standard for AOSP images. Any deployment beyond benchmarks would additionally require an allowlisted command surface, explicit user consent for destructive or privacy-sensitive commands, and preferably on-device inference. Human participants took part only as annotators and reviewers of oracle solutions, task realism, and failure labels; they saw no personal data. The Beyond-GUI Intent Suite is LLM-generated and human-verified, and contains only synthetic fixtures. We use AndroidWorld, MobileWorld, and all models in accordance with their licenses and terms of use.

REPRODUCIBILITY STATEMENT

We release all agents, prompts, tools, oracle solutions, the Beyond-GUI Intent Suite, and our evaluation infrastructure at `https://anonymous.4open.science/r/CLI_Agent-47D8`. The system prompts for all harnesses are described in Appendix A.1 and released verbatim; the tool suite is specified in Appendix A.2 (Table 4); the MobileCLI harness is detailed in Appendix A.4; and the oracle construction and verification protocol, including the lists of non-CLI-solvable tasks, is given in Appendix A.5 (Tables 7 and 8). Leaderboard numbers come from the snapshots cited in Appendix C.1. Because several agents rely on hosted model APIs whose outputs are not fully deterministic, we also release all trajectories and log the exact model identifiers used.

REFERENCES

Anthropic. Claude Code. `https://www.anthropic.com/claude-code`, 2025. Accessed: 2026-05-01.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495, 2024.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=fAjbYBmonr.

Yuxiang Chai, Shunye Tang, Han Xiao, Weifeng Lin, Hanhao Li, Jiayu Zhang, Liang Liu, Pengxiang Zhao, Guangyi Liu, Guozhi Wang, et al. A3: Android agent arena for mobile gui agents with essential-state procedural evaluation. *arXiv preprint arXiv:2501.01149*, 2025.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, et al. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. V-droid: Advancing mobile gui agent through generative verifiers. 2026.

Hongchao Du, Shangyu Wu, Qiao Li, Riwei Pan, Jinheng Li, Youcheng Sun, and Chun Jason Xue. Clawmobile: Rethinking smartphone-native agentic systems. In *Proceedings of the Sixth European Workshop on Machine Learning and Systems*, pp. 370–376, 2026.

Google. Android debug bridge (ADB). https://developer.android.com/tools/adb, 2025a. Accessed: 2026-04-20.

Google. dumpsys system diagnostics. https://developer.android.com/tools/dumpsys, 2025b. Accessed: 2026-04-20.

Li Gu, Zihuan Jiang, Zhixiang Chi, Huan Liu, Ziqiang Wang, Yuanhao Yu, Glen Berseth, and Yang Wang. Generalization in online reinforcement learning for mobile agents. *arXiv preprint arXiv:2603.07432*, 2026.

Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, et al. Ui-venus technical report: Building high-performance ui agents with rft. *arXiv preprint arXiv:2508.10833*, 2025.

Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun Peter Chen, and Yang Wang. Agent-sama: State-aware mobile assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29459–29467, 2026.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14281–14290, 2024.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Trans. Softw. Eng. Methodol.*, February 2026. ISSN 1049-331X. doi: 10.1145/3796519. URL https://doi.org/10.1145/3796519. Just Accepted.

Xueyu Hu, Tao Xiong, Biao Yi, Zishu Wei, Ruixuan Xiao, Yurun Chen, Jiasheng Ye, Meiling Tao, Xiangxin Zhou, Ziyu Zhao, et al. Os agents: A survey on mllm-based agents for computer, phone and browser use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7436–7465, 2025.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025.

Juyong Lee, Woogyeol Jin, and Kimin Lee. Improving proficiency and efficiency of android gui agents via self-generating tool actions. *arXiv preprint arXiv:2609.06792*, 2026a.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026b.

Chenxin Li, Zhengyao Fang, Zhengyang Tang, Pengyuan Lyu, Xingran Zhou, Xin Lai, Fei Tang, Liang Wu, Yiduo Guo, Weinong Wang, et al. Phoneharness: Harnessing phone-use agents through mixed gui, cli, and tool actions. *arXiv preprint arXiv:2606.14832*, 2026.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on ui control agents. *Advances in Neural Information Processing Systems*, 37:92130–92154, 2024.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for gui visual agent. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 19498–19508, 2025.

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Yue Han, Shuai Ren, et al. Llm-powered gui agents in phone automation: Surveying progress and prospects. *arXiv preprint arXiv:2504.19838*, 2025.

Ziwei Liu, Borui Kang, Hangjie Yuan, Zixiang Zhao, Wei Li, Yifan Zhu, and Tao Feng. Continual GUI agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=7WeA5TAjHK.

Ryan Lopopolo. Harness engineering: leveraging codex in an agent-first world. *OpenAI engineering note*, 2026.

Renze Lou, Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Suman Nath, Wenpeng Yin, and Jianfeng Gao. The tool illusion: Rethinking tool use in web agents. *arXiv preprint arXiv:2604.03465*, 2026.

Kevin Meng, Vincent Huang, Jacob Steinhardt, and Sarah Schwettmann. Introducing docent. https://transluce.org/introducing-docent, March 2025.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench:

Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

MiniMax. MiniMax M2.7: Early echoes of self-evolution. https://www.minimax.io/news/minimax-m27-en, March 2026. Accessed: 2026-05-01.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William E Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Kenji Toyama, Robert James Berry, Divya Tyamagundlu, Timothy P Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=il5yUQsrjC.

Yucheng Shi, Wenhao Yu, Zaitang Li, Yonglin Wang, Hongming Zhang, Ninghao Liu, Haitao Mi, and Dong Yu. Mobilegui-rl: Advancing mobile gui agent through reinforcement learning in online environment. *arXiv preprint arXiv:2507.05720*, 2025.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. CoAct-1: Computer-using multi-agent system with coding actions. In *The Fourteenth International Conference on Learning Representations*, 2026.

Yueqi Song, Frank F. Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: API-based web agents. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11066–11085, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.577. URL https://aclanthology.org/2025.findings-acl.577/.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026a.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, et al. Ui-venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*, 2026b.

Muxin Tian, Zhe Wang, Blair Yang, Zhenwei Tang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. Swe-bench mobile: Can large language model agents develop industry-level mobile applications? *arXiv preprint arXiv:2602.09540*, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems*, 37:2686–2710, 2024a.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024b.

Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *arXiv preprint arXiv:2406.08184*, 2024c.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024d.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th annual international conference on Mobile computing and networking*, pp. 543–557, 2024.

Biao Wu, Yanda Li, Zhiwei Zhang, Yunchao Wei, Meng Fang, and Ling Chen. Foundations and recent trends in multimodal mobile agents: A survey. *arXiv preprint arXiv:2411.02006*, 2024.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: Foundation action model for generalist gui agents. In *International Conference on Learning Representations*, volume 2025, pp. 5090–5108, 2025.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, et al. Mobile-agent-v3. 5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*, 2026.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Wenyi Zhao, and Yuxiao Dong. Mobilerl: Online agentic reinforcement learning for mobile gui agents. *arXiv preprint arXiv:2509.18119*, 2025a.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2144–2166, 2025b.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.

Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, et al. Step-gui technical report. *arXiv preprint arXiv:2512.15431*, 2025.

Jianwei Yang, Reuben Tan, Qianhui Wu, Ruijie Zheng, Baolin Peng, Yongyuan Liang, Yu Gu, Mu Cai, Seonghyeon Ye, Joel Jang, et al. Magma: A foundation model for multimodal ai agents. In *Proceedings of the computer vision and pattern recognition conference*, pp. 14203–14214, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Zhenquan Yao, Zitong Huang, Yihan Zeng, Jianhua Han, Hang Xu, Chun-Mei Feng, Jianwei Ma, and Wangmeng Zuo. Cgl: Advancing continual gui learning via reinforcement fine-tuning. *arXiv preprint arXiv:2603.02951*, 2026.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pp. 240–255. Springer, 2024.

Chaoyun Zhang, Shilin He, Liqun Li, Si Qin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. API agents vs. GUI agents: Divergence and convergence. In *ICML 2025 Workshop on Computer Use Agents*, 2025a. URL https://openreview.net/forum?id=XuThXhPSQR.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pp. 1–20, 2025b.

Pengxiang Zhao, Guangyi Liu, Yaozhen Liang, Weiqing He, Zhengxi Lu, WenHao Wang, Yuehao Huang, Yuxiang Chai, Zhaolu Kang, Yaxuan Guo, et al. Mas-bench: A unified benchmark for shortcut-augmented hybrid mobile gui agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6960–6985, 2026.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, et al. Mai-ui technical report: Real-world centric foundation gui agents. *arXiv preprint arXiv:2512.22047*, 2025.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, et al. Qwen-ui-agent technical report: Toward next-generation real-world centric foundation gui agents. *arXiv preprint arXiv:2607.28227*, 2026a.

Xiao Zhou, Siyue Zhang, Yilun Zhao, Jinbiao Wei, Tingyu Song, Arman Cohan, and Chen Zhao. Gui vs. cli: Execution bottlenecks in screen-only and skill-mediated computer-use agents. *arXiv preprint arXiv:2606.24551*, 2026b.

A Design and Implementations on CLI Agent

For each benchmark (AndroidWorld, MobileWorld), we evaluate the CLI paradigm under three off-the-shelf coding-agent harnesses (Claude Code, Terminus-2, and mini-SWE-agent) and one purpose-built harness (MobileCLI). This appendix details the design and implementation of each: the system-prompt core shared by the off-the-shelf harnesses (Appendix A.1), the tool specifications (Appendix A.2), the access model of each paradigm (Appendix A.3), the purpose-built MobileCLI harness (Appendix A.4), and the oracle construction protocol (Appendix A.5).

A.1 Off-the-shelf coding agents

Each of the three harnesses uses a separate system prompt for each of the two benchmarks, giving six distinct prompts in total. Each prompt comes in two variants, with and without the typed tools (Appendix A.2). We release all prompts and their variants verbatim with our code at `https://anonymous.4open.science/r/CLI_Agent-47D8`.

Anatomy of a prompt

All six prompts share a common skeleton, organized around the four categories of guidance described in Section 3:

1. **Four-phase interaction cycle:** discover relevant data, inspect existing state, act through the terminal interface, and verify the result. The cycle appears verbatim under the *Approach* heading in each prompt and is preceded by three preflight questions ("what has been observed," "what is still an assumption," "what is the most likely failure mode"). Where a write touches a cached app process, the Verify phase concludes with a force-stop sync step so the app re-reads on next launch.
2. **Prioritised hierarchy of mechanisms for modifying device state.** A ranked list of write paths to try in order: content providers first (so derived columns and change notifications fire), then system service commands (`cmd <service>`, `service call`, `am broadcast`) for live-service effects, then direct SQLite or file writes under `/data/data/<pkg>/` for app-private state with no provider. On MobileWorld the same hierarchy is restated as three writing surfaces (on-device, container backend, network backend) so the agent picks the layer by where the consumer reads.
3. **Efficiency strategies.** Two rules: batch related probes into a single shell call rather than one probe per turn, and respect a small probe budget — if 2–3 probes have not surfaced the answer, switch tactic rather than continuing with forbidden time-sinks (extracting APKs, full `dumpsys package`, recursive `find /`). Together they cap exploration cost while keeping the agent deliberate per call.
4. **Platform-specific patterns for file synchronisation and data persistence.** A short list of non-obvious Android behaviours: media-scanner broadcast after writes to shared storage, the content-provider self-notify semantics that make manual broadcasts unnecessary for canonical providers, and the recovery procedure for a corrupt SQLite file (remove the DB plus its WAL/SHM siblings, then recreate the schema). On MobileWorld this category also covers the device file-system layout, database discovery patterns, and the half-open range-query convention for UTC timestamps.

Around these four behavioural categories each prompt also carries a small set of structural blocks: a one-sentence agent role; a fenced *command space* listing the callable verbs and argument shapes (the AW listing names the four core wrappers from Appendix A.2 plus `adb shell` and `finish`; the MW listing adds the eight MW-only verbs); an explicit no-screen constraint forbidding screenshots, UI hierarchy dumps, taps/swipes/keyevents, and `adb pull` / `push` / `root`; a shell-escaping note recommending stdin pipes to avoid double-escaping inside `adb shell`; and a final response-format block whose contents depend on the harness (see below).

No task-specific or app-specific information that could leak to CLI agents. Every category above describes either (a) a generic device-control capability (`sql`, `read-file`, `adb shell`, `content query`, ...), (b) a generic Android platform convention that holds for any

app (`/data/data/<pkg>/`, content-provider notify semantics, half-open range queries on UTC timestamps), or (c) a generic agent-behaviour principle (never assume, copy task values verbatim, stop on completion). The task instruction itself enters through a placeholder (`{{ task }}`, `%INSTRUCTION%`, or the harness's post-system message) and the prompt body never names a specific benchmark task, application package, file path read by a verifier, or expected answer.

ACROSS-HARNESS DIFFERENCES

Within a benchmark, the three harness prompts are *almost identical* in content: the command list, constraint, approach cycle, principles, shell-escaping rule, Android patterns, and rules are word-for-word the same. The differences are confined to two surface dimensions:

1. **Output structure.** Each harness imposes a different response shape. The Claude Code SDK harness uses native tool calls and therefore needs no response-format block — the agent invokes the shim binaries directly via the SDK's Bash tool. The mini-SWE-agent harness asks for a free-form `THOUGHT:` line followed by exactly one fenced ```` ```bash ```` command. The Terminus-2 harness asks for a single JSON object `{"analysis": "...", "plan": "...", "command": "..."}`.
2. **Section ordering and partitioning.** The Claude SDK prompt is a single block. The mini-SWE-agent prompt is split into a system template (role + command space + format example) and an instance template (constraint + approach + principles + rules, rendered per turn with the task interpolated) following its original design. The Terminus-2 prompt is a single template that places *Response Format* and *Shell Escaping* immediately after the command list and defers the *Constraint* block to after them; the AndroidWorld instance also moves *Android Platform Knowledge* before the approach cycle. These reorderings are cosmetic — no behavioural content is added or removed.

An abridged view of the AndroidWorld prompt of the Claude Code SDK harness (the *bash+typed-tools* variant) follows; step and principle descriptions are elided, and the full prompt text is in the repository.

````
You are an Android automation agent. You control an Android device by
↪  writing shell scripts that interact with ADB through a CLI wrapper.

## CLI Tools
```bash
sql <db_path> "<SQL>"
read-file <device_path>
write-file <device_path> '<content>'
find-files <directory> "<pattern>"
adb shell <command>
finish --status complete --description "<result>"
```

## Constraint
Do NOT observe or manipulate the screen -- no screenshots, no UI
↪  hierarchy dumps, no tap/swipe/keyevent input, no screen
↪  coordinates, no adb pull/push/root. All other ADB shell
↪  capabilities remain available.

## Approach  (five steps; descriptions elided)
1. Discover [...]
2. Inspect [...]
3. Act [...]
4. Verify [...]
5. Sync [...]

## Principles  (eight rules; titles only)
1. Never assume [...]
2. Ground in reality [...]
3. Use exact task values [...]
4. Find the destination, don't invent it [...]
````

```
5. Probe budget [...]
6. Stop when done [...]
7. Verify through the surface a consumer reads from [...]
8. Discover before guessing a verb [...]

## Rules
1. For information-retrieval tasks, `--description` in `finish` IS your
↪  answer. Give ONLY what was asked -- no extra commentary.
2. ALWAYS call `finish` when done with a meaningful `--description`.

## Shell Escaping / Android-Specific Patterns  [omitted; see the
↪  released prompt]
```

A.2 TOOL SPECIFICATIONS

Each benchmark exposes a fixed CLI tool suite to the agent; the suite is identical across the three harnesses (Claude Code SDK, mini-SWE-agent, Terminus-2). Following the framing in Section 3, all tools are *general-purpose wrappers around raw ADB shell commands* that lift a raw shell verb into a higher-level operation: AndroidWorld receives four tools; MobileWorld extends the suite to twelve by adding eight more that reach the parts of the environment a device shell alone cannot. Table 4 lists every tool, marks the benchmark(s) on which it is active, and summarises what it does.

Core four (AW + MW). The four wrappers shipped with both benchmarks — `sql`, `read-file`, `write-file`, `find-files` — each lift a single `adb shell` verb into a higher-level operation: SQL queries, file I/O, and file search. The wrapper layer handles the multi-layer shell parses, the base64 round-tripping for quotes / unicode / binary content, and the glob re-quoting for device-side `find`. Anything an agent can do through these four it could in principle also do through a sufficiently careful `adb shell` invocation; the wrapper just removes a class of quoting bugs.

MW extensions (eight more tools). MobileWorld presents a more challenging setting with backend services, structured data, and networked APIs, so eight additional wrappers are exposed there:

- *Android platform IPC* — `content` and `intent` lift the standard content-provider and `am start`/`broadcast` surfaces into typed `-bind` / `-extra` forms.
- *Structured-data parsing* — `json-read` and `json-write` add a structured-JSON read/write path on top of `adb shell cat` and `write-file` so the agent does not have to compose jq inside a nested shell.
- *Networked APIs* — `http` reaches MobileWorld's in-environment HTTP endpoints by routing requests through the correct network namespace, so the agent's `localhost:<port>` resolves to the publishing container rather than the runner's loopback.
- *Backend services* — `pg`, `backend-exec`, and `service-status` reach into the nested Docker-in-Docker daemon where MobileWorld's backend services (Mastodon, Mattermost, ...) live; without them the agent cannot interact with the backend half of MobileWorld at all.

AndroidWorld has no backend services and no JSON-as-app-state contract, so these eight verbs are not exposed there.

The raw surface and termination signal. Two additional verbs sit outside the wrapper count. The raw `adb shell <cmd>` pass-through is always available on both benchmarks as the underlying surface the wrappers sit on top of, used as an escape hatch when no typed verb fits. The harness termination signal `finish -status complete -description "<answer>"` marks the episode terminated and records the agent's answer for IR tasks; it is a signalling primitive rather than a device or backend operation.

No task- or app-specific information. Consistent with the system-prompt claim in Appendix A.1, the tool suite contains no task-specific or app-specific information that could leak to CLI agents. Every verb in Table 4 maps to a generic systems primitive — shell, SQLite, file system, glob,

Table 4: CLI tool suite. AW = AndroidWorld, MW = MobileWorld; ✓ indicates the verb is callable by the agent on that benchmark.

Tool	AW	MW	What it does
Core file / SQL wrappers (AW shipped suite)			
`sql`	✓	✓	Run SQL on a device SQLite DB; pipes the query via base64+stdin into `adb shell sqlite3` so quotes / unicode survive nested parses.
`read-file`	✓	✓	Read a device file via base64-round-tripped `adb shell`. On MW, `.pdf` paths are additionally extracted host-side through a `pdftotext`/PyMuPDF/pypdf/strings fallback chain.
`write-file`	✓	✓	Write content to a device path via base64-encoded `adb shell`; `-append` supported.
`find-files`	✓	✓	`adb shell find <dir> -name <glob> -maxdepth N`, with the glob re-quoted so the device shell does not pre-expand it.
MW extensions — Android platform IPC			
`content`		✓	Wrap the `adb shell content query/ insert/ update/ delete` verbs with typed `-bind` fields and standard `-projection` / `-where` / `-sort`.
`intent`		✓	Wrap `adb shell am start/broadcast` with typed extras (string, int, bool, long, float, uri).
MW extensions — structured-data parsing			
`json-read`		✓	Parse a JSON file on the device with a jq-style path expression (`.key[0].sub`).
`json-write`		✓	Validate and write structured JSON to a device path; `-merge` deep-updates a top-level dict.
MW extensions — networked APIs			
`http`		✓	HTTP client with namespace-aware routing: rewrites admin loopback ports to the broker URL, runs `curl` inside the publishing container for other loopback URLs, and uses `urllib` for external URLs.
MW extensions — backend services			
`pg`		✓	PostgreSQL client into a nested backend container; auto-discovers the container by grep on `docker ps`, then pipes SQL via stdin to `psql`.
`backend-exec`		✓	Run an arbitrary shell command inside a nested backend container (same grep+stdin pattern as `pg`).
`service-status`		✓	Read-only `docker ps` against the nested Docker-in-Docker daemon; authoritative list of running backend containers.

content provider, intent, HTTP, JSON, `docker ps`, `psql` — and no verb names a benchmark concept. The wrappers treat all agent-supplied arguments as opaque strings; they do not parse the task instruction, do not dispatch on the task name, do not read the verifier source, and do not return verifier internals. The same binary serves every task in the benchmark. Hazardous verbs that could exfiltrate state into or out of the simulated device (`adb pull`, `adb push`, `adb root`) are blocked by a benchmark-wide deny pattern.

A.3 Comparison between the GUI and CLI paradigms

Table 5 compares what each paradigm reads, at what privilege level, through which interfaces, and whether that access is available on an unmodified consumer device. Table 6 classifies each tool in our harness by the same access surface. Its *Locus* column also defines the split in Appendix C.1: a MobileWorld task counts as backend-assisted if its oracle solution uses any backend tool.

Table 5: Access model of the two paradigms. The asymmetry is twofold: on-device CLI access is more privileged than user-level GUI control and requires provisioning, while the backend tools are benchmark infrastructure rather than a consumer-device capability.

Paradigm	What it reads	Privilege level	Interfaces	Real device?
GUI	Rendered screen and accessibility tree	User-level, via accessibility service	Accessibility API; screenshot	Yes
CLI, on-device	App databases, files, content providers, system settings, and hidden device state (package and permission info, service dumps)	Shell-user privilege via ADB; no permission prompts; elevated relative to ordinary apps (emulator images additionally permit root)	`adb shell`: `content`, `settings`, `pm`, `dumpsys`; SQLite; filesystem	Only with developer mode or host provisioning
CLI, backend (MobileWorld)	Backend service databases, container state, running-service listings, and HTTP endpoint data	Infrastructure-level, outside the device	PostgreSQL; container shell; service status; HTTP endpoints	No (benchmark infrastructure)

Table 6: Per-tool access classification for the CLI harness. Backend tools exist only on MobileWorld; they define its backend-assisted split (Appendix C.1).

Tool	Locus	Access mechanism	Consumer device?
AndroidWorld			
`bash`	on-device	`adb shell`	With provisioning
`sql`	on-device	SQLite on app databases; bypasses provider authority	With provisioning
`read-file`	on-device	Filesystem read; bypasses SAF	With provisioning
`write-file`	on-device	Filesystem write; bypasses SAF	With provisioning
`find-files`	on-device	Filesystem traversal	With provisioning
MobileWorld: the five tools above, plus			
`content`	on-device	Content-provider queries and updates as the shell user	With provisioning
`intent`	on-device	Activity starts and broadcasts as the shell user	With provisioning
`json-read`	on-device	Filesystem read of JSON app state	With provisioning
`json-write`	on-device	Filesystem write of JSON app state; bypasses SAF	With provisioning
`http`	backend	HTTP requests routed to MobileWorld's service endpoints	No (benchmark infrastructure)
`pg`	backend	PostgreSQL client in a backend container	No (benchmark infrastructure)
`backend-exec`	backend	Shell inside a backend container	No (benchmark infrastructure)
`service-status`	backend	Listing of running backend containers	No (benchmark infrastructure)

A.4 MOBILECLI: A PURPOSE-BUILT CLI HARNESS

Design. MobileCLI is a bash-only harness tuned and evaluated with Claude Opus 4.7: the agent emits shell commands and receives output, with none of the typed wrappers of Appendix A.2 and the same no-screen constraint as every other CLI configuration. Three components distinguish it from the off-the-shelf harnesses of Appendix A.1 — device-state context, mobile-specific error feedback, and a budget-aware control loop:

1. **Device-state context.** Before the first turn the harness runs one probe script and summarizes the device states into an *environment card* placed in the task message: the device shell identity; the date, time zone, and SDK level; the installed third-party packages, and, for each, how much state it keeps — databases, files, shared-preference files and external files,

counted by a probe over its data directories. The card also lists the backend containers by name, if any. The card holds counts and names only — never a setting's value, a file name or a row — and aids the discovery and exploration of the device. An example of the card is shown below.

2. **Mobile-specific error feedback.** Device `stderr` is classified by generic Android and SQLite signatures and returned under a one-line header that names the class, followed by a short note on what to do next. The classes are `[denied: ...]`, `[provider: ...]` (a silently ignored provider write), `[sqlite: ...]` (schema, lock or statement), `[binder: ...]`, `[usage: ...]`, `[not found: ...]` and `[timeout: ...]`. Both the card and the classifier are benchmark-independent and work on any Android device reachable over ADB.
3. **Budget-aware control loop.** The harness reminds the agent to balance the three phases of exploration, action, and verification, and to finish within the budget. Mid-way and near-budget checks are inserted to remind the agent to end exploration and finish within the harness-set budget. Two failure patterns are caught by the harness itself: a write identical to one already applied is returned without execution, and a verdict of *infeasible* is accepted only after at least three device commands. An agent that force-stopped an app without relaunching it is challenged once before its `finish` is accepted.

```
shell: uid=0(root) | Thu Oct 16 12:00:28 GMT 2025 | tz=GMT | sdk=34
third-party packages -- how much state each keeps (db=databases,
↪  files=private files, prefs=preference files, ext=external files):
  at.tomtasche.reader: db(0) files(4) prefs(2) ext(0)
  com.android.adbkeyboard: db(0) files(0) prefs(0) ext(0)
  com.gmailclone: db(0) files(2) prefs(1) ext(2)
  com.mattermost.rnbeta: db(0) files(6) prefs(8) ext(0)
  com.testmall.app: db(1) files(3) prefs(9) ext(0)
  gallery.photomanager.picturegalleryapp.imagegallery: db(5) files(13)
  ↪  prefs(30) ext(1)
  org.fossify.calendar: db(1) files(2) prefs(1) ext(0)
  org.joinmastodon.android.mastodon: db(3) files(0) prefs(8) ext(0)
shared storage: Alarms(0) Android(3) Audiobooks(0) DCIM(1) Documents(8)
↪  Download(30) Movies(0) Music(0) Notifications(0) Pictures(4)
↪  Podcasts(0) Recordings(0) Ringtones(0)
backend containers: mastodon-docker-db-1 mastodon-docker-nginx-1
↪  mastodon-docker-redis-1 mastodon-docker-sidekiq-1
↪  mastodon-docker-streaming-1 mastodon-docker-web-1
```

Tuning on MobileWorld, evaluation on AndroidWorld. MobileCLI was tuned on MobileWorld with Claude Opus 4.7 and then frozen and run on AndroidWorld without further tuning, so AndroidWorld measures transfer to an unseen benchmark. No component contains task-specific or app-specific information. The harness development targeted three patterns in Claude Code's MobileWorld runs: the imbalance between exploration and exploitation, the misreading of mobile-specific (or hidden) error messages, and incomplete verification. The prompt is built from the core principles of Appendix A.1, adapted to the MobileCLI harness, and was audited against the task definitions before the runs we report.

Harness differences between MobileWorld and AndroidWorld. The two benchmark configurations differ only in the action space and the prompt text that describes it; every other component is shared. For AndroidWorld we removed the backend-related text from the MobileWorld prompt — the host-shell action and its guide, the section on backend surfaces, the backend containers on the environment card, and three one-clause backend mentions — and kept everything else the same.

A.5 Oracle construction protocol

This appendix documents how we built the CLI-solvable subset of each benchmark and the ground-truth trajectories used as reference solutions through human–LLM collaboration. The subset counts cited in Section 3 are **88.8% (103/116)** on AndroidWorld and **86.3% (101/117)** on MobileWorld;

the remaining 13 tasks on AW and 16 tasks on MW involve interactions outside the CLI paradigm's reach — interacting with visual content (e.g., transcribing receipts from photos, free-hand drawing) or capturing multimodal data (e.g., camera photos or audio recording), plus a small number of UI-panel-only data sources on MW (Google Maps detail panels, in-app cart totals) — and are enumerated at the end of this section.

(1) THE TWO ACCEPTANCE CRITERIA

A task is admitted into the CLI-solvable subset only if its candidate trajectory satisfies *both*:

Solvability. The candidate trajectory must pass the benchmark's own rule-based verifier when replayed against a freshly reset container at the canonical seed (seed 7 for AW; per-task default for MW). Solvability is checked by resetting the container to the canonical initial state, dispatching each step in the candidate trajectory through the production CLI tool surface (the same wrappers the scoring agent uses; see Appendix A.2), issuing `finish`, and reading the binary reward from the container's verifier endpoint. We re-run this check inside a validation pipeline that dispatches every step in the published ground-truth reference against a live broker and fails on any non-zero return code or final score $\neq 1$.

Integrity. Passing the verifier is necessary but not sufficient. We additionally require:

1. *No verifier-internals leakage.* The trajectory must be producible from device state and the task goal alone, without reading the verifier source, its private fixtures (e.g. parameter-bearing attributes on the task class), or any gold-answer file.
2. *No hardcoded answers.* Numeric and string answers for information-retrieval tasks must be *computed* from data the verifier itself reads, not hardcoded as constants in the trajectory.

(2) THE ORACLE-AGENT LOOP

We treat oracle construction itself as a human–LLM collaboration: a separate *oracle-agent* (Claude Code, running on the host) is given a far richer context than the scoring agent and proposes, for each task, a condensed trajectory that a human reviewer then validates. Each task is attempted up to five times with verifier feedback; the loop runs in five phases per attempt.

Step 1 — Setup. The oracle-agent runs on the host, outside the per-task evaluation container. It is granted read access to artifacts the scoring agent never sees: the full body of past agent trajectories on this task (across models and harnesses), the system prompts those agents were run under, and the benchmark's emulator-design and tool-surface documentation. The oracle-agent's own action space is the same CLI tool suite from Appendix A.2, exposed through a single *oracle-entrypoint* that issues actions against a freshly reset container; everything else in the rich context is read-only research material.

Step 2 — Exploration. The oracle-agent first *reads* rather than acts. It scans the past trajectories for the task and labels each one: which final verifier-reward did it achieve, which steps did the past agent take, where did short trajectories diverge from long ones, what error messages recurred. Successful past trajectories surface a candidate *path skeleton*; failed past trajectories surface a *failure-mode list* (wrong URI, wrong table column, missing `am force-stop`, wrong timestamp unit, etc.).

Step 3 — Proposal. The oracle-agent condenses the path skeleton and the failure-mode list into the shortest CLI trajectory it believes will satisfy the verifier — replacing GUI-shaped detours with direct `sql` / `content` / `intent` verbs, collapsing verification rounds the scoring agents repeated unnecessarily, and naming exactly the data sources the answer should be computed from. The proposal is emitted as an ordered step list, not as free text.

Step 4 — Oracle-acting. The oracle-agent then dispatches the proposed trajectory through the oracle-entrypoint against a freshly reset container, end-to-end, and records the per-step output plus the final verifier reward. This is the first time the proposed steps actually touch the device; the exploration and proposal phases never modify state.

Step 5 — Human-in-the-loop, one reviewer per round. A human reviewer reads the full round output (proposal + per-step actuation + final reward) and renders one of two verdicts:

- **Invalid.** The verifier rejected the trajectory, or the trajectory was incomplete, or the reviewer disagrees with the approach. The reviewer gives *general advice only* — "you skipped the force-stop", "the Mattermost message has to be sent through the API, not the DB" — and never reveals verifier internals, expected answer values, or fixture file names. The advice is appended to the rich-context bundle and the loop re-enters Step 2 with the next round.
- **Valid.** The verifier accepted the trajectory. The reviewer then runs the integrity audit for verifier-internals leakage and hardcoded answers. If the trajectory clears both checks the round closes and the trajectory is committed to the ground-truth reference; if any check fails the round is re-opened with the violating step flagged for rewrite, or the task is escalated to the manual-verification stage below.

The five-phase loop iterates up to five times per task until either a valid+clean trajectory is committed or the task is judged un-solvable under the integrity rules and demoted to the GUI-only list.

(3) Manual verification

Each task that exits the oracle-agent loop with a committed trajectory is then manually verified independently by three external professionals. Each professional re-runs the solvability check end-to-end against a fresh broker pool and, separately, audits the integrity of every step under the rejection signals below.

Reconciliation rule. A task is admitted into the CLI-solvable subset only if all three professionals mark it pass on both solvability and integrity. Any single rejection demotes the task to the GUI-only list; ties are resolved by re-running the trajectory live with all three professionals present, and the demotion stands if the live re-run does not resolve the dispute.

(4) Not-CLI-solvable task lists

The 13 tasks on AW and 16 tasks on MW listed below fall outside the CLI paradigm's reach: either they require interacting with visual content (transcribing receipts from photos, free-hand drawing, image-content classification) or they require capturing multimodal data (camera photos, audio recording); MW additionally contributes a handful of cases where the required data point is only rendered in an in-app panel (Google Maps drive-/walk-time, business detail fields, shopping-cart totals) with no public API path. We list both benchmarks separately — AndroidWorld (Table 7) and MobileWorld (Table 8) — giving the task name and the precise reason CLI access cannot satisfy the verifier.

AndroidWorld (AW). The 13 tasks excluded from the AW CLI-solvable subset are listed in Table 7; all require either vision (OCR or image content) or a UI-side capture path (camera shutter, audio recorder, drawing canvas).

MobileWorld (MW). The 16 tasks excluded from the MW CLI-solvable subset are listed in Table 8; they include the same vision and multimodal-capture categories as AW, plus two MW-specific patterns that AW does not surface: (i) Google Maps panel-only labels (business phone, neighbour name, walk-/drive-time) that have no public API path, and (ii) e-commerce cart fields (item list, cart total) that are only rendered in the in-app cart panel. We treat a task as GUI-only if either CLI execution cannot in principle satisfy the verifier, or the verifier nominally accepts a CLI path but the correct answer is determined by image content that the verifier does not enforce (e.g. a filename-suffix check that happens to align with the visual content).

B Beyond-GUI Intent Suite — Construction and Extended Results

This appendix documents the Beyond-GUI Intent Suite introduced in Section 4: the full per-category task list (§B.1), the seed-driven randomization mechanism and the two rule-based verifier types

Table 7: AndroidWorld — 13 GUI-only tasks (out of 116; CLI-solvable subset = 103).

Task	Why CLI cannot solve it
Capturing new media through a device UI	
`AudioRecorderRecordAudio`	Requires audio capture through the recorder UI; no programmatic shutter.
`AudioRecorderRecordAudio WithFileName`	Same as above, plus naming requires the recorder save dialog.
`CameraTakePhoto`	Requires capturing a real photo through the camera UI.
`CameraTakeVideo`	Camera video capture through the UI shutter.
Interactive UI with no CLI surface	
`ClockTimerEntry`	Timer-entry UI has no programmatic equivalent for "set but do not start".
`BrowserDraw`	Drawing in a Chrome canvas with three colours.
`BrowserMaze`	Navigating a maze widget in Chrome via direction buttons.
`BrowserMultiply`	Clicking a button five times in Chrome and reading the displayed numbers; also needs OCR.
`SimpleDrawPro CreateDrawing`	Requires producing pixels in a drawing app and naming the export.
Content legible only to OCR or vision	
`MarkorTranscribeReceipt`	Transcribe `receipt.png` content as CSV.
`ExpenseAddMultiple FromGallery`	Transcribe expenses from `expenses.jpg`.
`MarkorTranscribeVideo`	Transcribe per-frame text from a VLC video.
`RecipeAddMultiple RecipesFromImage`	Transcribe recipes from `recipes.jpg`.

(§B.2), the oracle-construction protocol (§B.3), the realism rubric and three-annotator agreement (§B.4), and the semantic similarity to AndroidWorld (§B.5).

B.1 Full task list

We list all 45 tasks by category, showing the natural-language instruction template each task ships with. Per task, the seed varies the on-device fixture (data values, row counts, file sets, granted permissions, etc.) and, where present, the angle-bracket placeholder fillers (e.g., `<keyword>`). Section B.2 describes the randomization contract and what specifically varies seed-to-seed for each category.

Bulk operations (10 tasks).

- Delete all `.tmp` files in the Downloads folder.
- Rename all files starting with `Screenshot_` in the Pictures folder to the format `YYYYMMDD_HHMMSS.png` based on each file's modification time.
- Move all files larger than 50 MB in the Download folder to the Archive folder; create the Archive folder if it does not exist.
- Append the footer "`---\nGenerated by AutoBot`" to every `.md` file in the `Notes` folder in Markor.
- Change all *Food* entries in Pro Expense to *Entertainment*.
- Change all overdue tasks in the Tasks app to *High* priority.
- Delete all events in Simple Calendar Pro whose title contains the word *test* (case-insensitive).
- Delete duplicate calendar events in Simple Calendar Pro (same title and same start time), keeping only one copy of each.
- Delete all `.apk` files in the Downloads folder.
- Delete all expenses in Pro Expense that are less than $1.00 (less than 100 cents).

Table 8: MobileWorld — 16 GUI-only tasks (out of 117; CLI-solvable subset = 101).

Task	Why CLI cannot solve it
Content legible only to OCR or vision	
`MattermostVisual InstructionResponseTask`	Contacts and shift times appear only in whiteboard photos posted to a Mattermost channel.
`DownloadSendReceiptTask`	The receipt total is readable only from the receipt image.
`ChangeWallpaperTask`	Identifying the sunflower photo among four pushed images.
`SharePhotosTask`	Identifying which gallery images show flowers.
`MattermostSendFileTask`	Identifying the birthday-cake image to upload.
`PhotoManagementTask`	Identifying food photos; the verifier's filename-suffix check only happens to align with the visual content.
`MastodonChangeHeaderTask`	Identifying the tiger photo; the file `tiger.jpg` only happens to be present.
Capturing new media through a device UI	
`TakeSelfieTask`	Requires the camera UI to capture an actual photo; no CLI shutter path.
(i) Google Maps panel-only labels, with no public API path	
`GoogleMapsAlibaba PhoneContactTask`	Business phone number, shown only in the Maps detail panel.
`GoogleMapsAlibaba SouthNeighborTask`	Neighbour name, shown only in the Maps detail panel.
`CheckConference LocationTask`	Walk time between two pins, shown only in the Maps UI.
`TextArrivalTimeTask`	Drive time between two cities, shown only in the Maps UI.
(ii) E-commerce cart fields rendered only in the in-app cart panel	
`CartManagementTask`	Cart item list; modifying items requires the cart UI, and no backend API exposes the operations the verifier checks.
`CheckCartPriceTask`	Cart total, shown only in the cart panel.
Mastodon workflows exposed only through the in-app UI	
`MastodonManage MultiListTask`	The list-management workflow.
`MastodonShare LocationTask`	Sharing a location in a post requires the in-app location picker.

Multi-condition filtering (10 tasks).

- List all contacts that have a birthday set but no phone number; output the contact names.
- List all contacts that have a phone number but no family name (first name only).
- List all Joplin notes that contain `<keyword_a>` but do NOT contain `<keyword_b>`; output the note titles.
- List all songs in Retro Music by artist `<artist>` that are longer than 4 minutes; output the song titles.
- Do all my calendar events this month have reminders set? If any are missing reminders, name them.
- List all SMS messages in the inbox that contain a URL (`http://` or `https://`); output the sender phone numbers.
- Are there any SMS messages in my inbox from numbers not in my contacts? If so, how many?
- List all zero-byte (empty) files in the Downloads folder; output the filenames.
- Which of my expenses are above my average spending? List them.
- What do I have planned on weekends this month? List all calendar events that fall on a Saturday or Sunday.

Aggregation and Top-K (10 tasks).

- Which 5 notes in Markor were modified most recently in the last 7 days? List the filenames.
- Which 3 phone numbers have the most SMS messages in your inbox? List their phone numbers.
- List all groups of contacts that share the same phone number; output each group with the shared phone number and the contact names.
- What are the top 3 expense categories by total amount this month in Pro Expense? Output the category names.
- How many suspected duplicate expenses are there in Pro Expense (same date, amount, and category)? Output the number of extras beyond the first of each group.
- What are the 5 highest-amount expenses in Pro Expense? List the expense names and amounts.
- What is the total distance of all activities this week in OpenTracks, and which activity covered the longest distance?
- Which activity in OpenTracks had the highest average speed? Output the activity name and its average speed.
- What are the 5 longest songs in Retro Music by duration? List their titles.
- What are the 5 largest files in the Downloads folder? List their filenames and sizes.

Cross-app workflows (9 tasks).

- Calculate this month's total expenses in Pro Expense, write the total (in dollars) to a Markor note `monthly_summary.md`, and create a Simple Calendar Pro event titled `Monthly Expense: $<total>` on the last day of this month.
- Which of my contacts texted me but I haven't replied to?
- Extract all phone numbers mentioned in Markor notes, then list the ones that are NOT in your contacts; output the numbers.
- Find all events in Simple Calendar Pro whose title contains `<keyword>`; create a Markor note `<keyword>_events.md` listing the event titles and dates, one per line.
- Export all contacts (name and phone number) to a Markor note `contacts_export.md`, one contact per line in the format `Name: phone_number`.
- Did anyone text me during my meetings yesterday? List the senders of all SMS received while a calendar event was active yesterday.
- Find all SMS messages in the inbox that contain the word *urgent* and create a task in the Tasks app for each one, using the SMS body as the task title.
- Summarize this week's OpenTracks activities (total distance in km and number of activities) and save it as a Markor note `weekly_stats`.
- Export all Joplin notes whose title contains *meeting* to Markor as separate notes.

Hidden device state (6 tasks).

- List all apps (by package or app name) that have been granted location permission on this device. If none, output *None*.
- List all apps that have been granted Camera permission on this device. If none, output *None*.
- What is my phone's current temperature? Is it overheating?
- What are the 3 most recently installed apps on the device? Output their package names.
- How long has the device been running since the last reboot? Output the uptime in hours and minutes.
- Which apps have recently accessed my location in the background?

B.2 RANDOMIZATION MECHANISM AND RULE-BASED REWARD

Every task is implemented as a Python class against the AndroidWorld `TaskEval` interface (the same contract used by the 116 stock AndroidWorld tasks). Each task class provides two methods:

- `initialize_task(env)` seeds all on-device fixtures from `random.Random(self.seed)` and binds any placeholder fillers in the goal text. The same `(task_id, seed)` reproduces the same fixture and the same instantiated goal sentence.
- `is_successful(env)` returns a binary reward (0.0 or 1.0) by reading device state and/or the agent's reported answer; the grader is deterministic, never invokes an LLM-judge, and never makes a network call.

We publish three evaluation seeds $\{7, 30, 1234\}$. What is stable across all three seeds: task ID, class name, category, goal-sentence template, and the verifier code itself; agents must solve the task class, not memorize values. What varies seed-to-seed differs by category, and is summarized below.

What the seed varies, by category. For *Bulk* tasks, the fixture is the set of files in a target directory or the set of rows in a target table; the seed picks which items exist, their attributes (sizes, timestamps, category and priority labels, expense amounts), and the size of the subset that must be acted on (e.g., how many files exceed the size threshold, how many expenses fall below $1.00). For *Filter* and *Aggregation* tasks, the seed populates the same kinds of fixtures and additionally computes the expected answer set on the host (the names that satisfy the predicate; the top-K; the duplicate groups), storing them as `self._expected`. For *Cross-app* tasks, the seed sets up both the source-app fixture and the post-action target-app expectation. For *Hidden device state* tasks, the seed configures device-level facts: which packages appear installed, which permissions are granted, the apparent uptime and temperature, and recent-install timestamps. Where the goal text contains angle-bracket tokens (`<keyword>`, `<artist>`, etc.), the seed additionally picks the substitution value before the goal is shown to the agent. Two end-to-end worked examples follow.

Two verifier function types. Every task bottoms out into one of two verifier shapes, picked by category. *Cross-app* tasks that both mutate device state and report a summary chain the two checks; we describe only the primitives below.

1. **State-check.** The grader re-reads device state after the agent finishes and compares it to the post-action expectation. The agent's natural-language output is ignored.

```
def is_successful(self, env) -> float:
    rows = adb_utils.issue_generic_request(
        "shell", "sqlite3 ...
        'SELECT ... FROM ... WHERE ...'",
        env)
    return 1.0 if matches_expected(rows)
    else 0.0
```

2. **Cache-match.** The grader inspects the agent's `FINISH(content=...)` payload. Every expected substring (computed in `initialize_task` from the seed) must appear; substring containment, case-sensitive, no whitespace normalization.

```
def is_successful(self, env) -> float:
    return 1.0 if all(
        s in env.interaction_cache
        for s in self._expected
    ) else 0.0
```

Example 1 *Goal:* "Delete all expenses in Pro Expense that are less than $1.00 (less than 100 cents)."

In `initialize_task`, `random.Random(seed)` populates the Pro Expense SQLite database `accounting.db` with a base set of ∼20 expenses uniformly sampled from $[100, 10{,}000]$ cents, plus a seed-derived count $k_{\text{small}} \in \{3, 4, 5\}$ of expenses with amounts in $[1, 99]$ cents. The goal text contains no placeholders; the fixture itself is what the seed varies. After the agent finishes,

`is_successful` runs `SELECT COUNT(*) FROM expense WHERE amount < 100` on `accounting.db` and returns 1.0 iff the count is zero.

Example 2 *Goal:* "Which of my expenses are above my average spending? List them."

`initialize_task` seeds the same expense-style fixture, then computes the expected answer set $E = \{\text{name} : \text{amount}(\text{name}) > \overline{\text{amount}}\}$ on the host and stores it in `self._expected`. The agent must produce a `FINISH` payload whose text contains every name in E as a substring; partial answers fail. As with the state-check example, E is recomputed on each seed.

B.3 Oracle construction

Each of the 45 tasks ships with a hand-verified golden-path oracle CLI command sequence that the rule-based verifier accepts. Oracle construction follows the same human–LLM collaborative protocol used for AndroidWorld and MobileWorld and therefore we do not repeat it here. See Appendix A.5 for the full protocol.

B.4 Realism rubric and annotator protocol

We retain a task only if its goal text reads like a request a real-world mobile user would actually make. Each of the three external mobile-product users independently rates every candidate task on a three-level rubric.

Rubric anchors.

- **Highly realistic** — phrasing matches how a non-technical user would describe the request in casual conversation. Examples: "*Delete every `.apk` file in Downloads*"; "*Top 3 expense categories this month*"; "*Which apps have been granted location permission?*".
- **Stretched** — the intent is plausible, but the surface form leaks engineering vocabulary or assumes more precision than a typical user would volunteer. Examples (paraphrased from excluded candidates): "*Which note has the highest character count?*"; "*Total download size in bytes*".
- **Contrived** — the task uses power-user, developer, or admin framing, or assumes the user already knows the implementation surface. Examples (excluded): "*Signal strength in dBm*"; "*Verify all expenses are categorized*"; "*SMS database storage size*".

Annotator instructions. Each annotator received the rubric anchors, a sheet with one representative quote per level, and the 56 candidate goal texts in random order. For every task, the annotator recorded a single label and a one-sentence justification. We did not show annotators the category, task ID, or implementation—only the surface goal text. The final inclusion rule is majority *highly realistic* across the three annotators; ties and minority-realistic tasks were discussed in a single reconciliation pass and dropped if consensus could not be reached.

Agreement. We report Cohen's κ averaged over the three annotator pairs (Table 9), consistent with the headline value reported in Section 4. Overall pairwise $\kappa = 0.91$; per-category figures are tightest on Bulk and Hidden State (the rubric levels are most distinguishable there) and loosest on Aggregation, where the line between "highly realistic" and "stretched" turns on whether the user would name the aggregation explicitly. The final yield is **45 of 56 candidates** retained.

B.5 Similarity to AndroidWorld

This subsection documents the methodology and density plot (Figure 3) for the additivity check referenced in Section 4. Each of the 45 Beyond-GUI instructions and each of the 116 AndroidWorld test instructions is embedded with `text-embedding-3-large`; goal-text fillers are instantiated using seed 7 before embedding so the comparison runs over concrete strings, not templates. We then form two similarity distributions, both reported as kernel density estimates with bandwidth chosen by Scott's rule:

Table 9: Three-annotator agreement on the realism rubric, reported as the mean Cohen's κ over the three annotator pairs. The overall value is the headline figure cited in Section 4.

Category	Pairwise κ	Retained / Candidates
Bulk	0.95	10 / 11
Filter	0.91	10 / 12
Aggregation	0.83	10 / 13
Cross-app	0.90	9 / 10
Hidden state	0.95	6 / 10
Overall	**0.91**	**45 / 56**

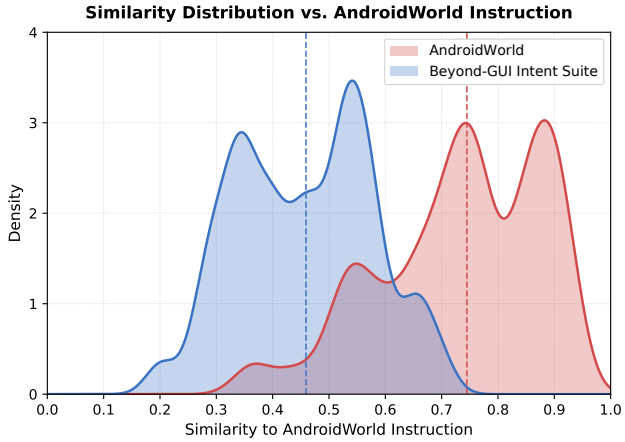


Figure 3: Semantic similarity between Beyond-GUI and AndroidWorld instructions. AndroidWorld is shown as a within-suite reference (each task's similarity to its nearest sibling), illustrating the benchmark's task-family regime. Beyond-GUI instructions lie distinctly to its left, with 0% exceeding similarity 0.7.

- **Beyond-GUI to AndroidWorld** (blue). For each Beyond-GUI instruction, the cosine similarity to its nearest neighbour in AndroidWorld. 45 samples, median ≈ 0.45.
- **AndroidWorld within-suite reference** (red). For each AndroidWorld instruction, the cosine similarity to its nearest sibling AndroidWorld instruction (i.e., k-NN with $k=1$, excluding self). 116 samples, median ≈ 0.75. This anchors what "in-family" looks like under the chosen embedding and metric.

The blue distribution is contained in $[0, 0.7]$ — 0% of Beyond-GUI instructions are as close to an AndroidWorld instruction as AndroidWorld instructions typically are to each other. The suite is additive in the embedding sense, not a redistribution of existing AndroidWorld coverage.

C Extended Results

This appendix expands the main-paper results along four axes: per-seed and leaderboard-extended versions of Table 1 on both AndroidWorld and MobileWorld (§C.1); matched-model pairs that hold the model API fixed and vary only the paradigm (§C.2); the tool suite, its ablation, and paired qualitative examples of where it changes agent behaviour (§C.3); and the cross-version generalization study behind Section 5.2 (§C.4).

C.1 Extended results on AndroidWorld and MobileWorld

Table 11 extends main-paper Table 1 with the full set of leaderboard-reported methods we are aware of on AndroidWorld (AW) and MobileWorld (MW). AW leaderboard numbers are sourced from the public GUI-Owl/MAI-UI benchmark comparison page; MW leaderboard numbers are sourced from

the official MobileWorld `leaderboard.json` snapshot at `https://tongyi-mai.github.io/MobileWorld/`, accessed 2026-05-24 (31 entries). Several methods are reported under slightly different display names across the two leaderboards (e.g. `Qwen3-VL-32B-Instruct` on AW vs. `Qwen3-VL-32B` on MW, `MAI-UI-8b` vs. `MAI-UI-8B`); we collapse these to a single row when the upstream identifier and release date match. **SR-CLI** and **Avg Step** columns are only populated for our reproduced GUI baselines and our CLI agents (mean (std) across three seeds, $\{7, 30, 1234\}$); leaderboard-only rows are populated where the authors release trajectories, from which we recompute SR-CLI and Steps under our own subset definition, and marked "–" where they release neither per-task verifier outputs nor step counts. "−" in the SR columns means the method does not currently appear on that benchmark's leaderboard.

Reproduction sanity. On AW, GUI-Owl-1.5-32B, MAI-UI-8B, and Qwen3-VL-32B fall within 2.6 points of the leaderboard (69.3 vs. 69.8, 68.1 vs. 70.7, 57.2 vs. 57.3). On MW, all four fall within 3.2 points (41.3 vs. 43.9, 27.4 vs. 27.5, 15.1 vs. 11.9, and 57.0 vs. 56.4 for Claude e2e). Claude e2e is the official MobileWorld end-to-end agent, run with Claude Opus 4.7 and with its prompt, action space, and actuation unchanged; it is screenshot-only, with one image per step. We port it to AW, where no official result exists. The other models are evaluated with their released pipelines.

Cross-benchmark difficulty drop. Every leaderboard method that reports both AW and MW drops substantially on MW — e.g. MAI-UI-235B-A22B from 76.7 to 39.7, GUI-Owl-1.5-32B-Instruct from 69.8 to 43.9, Qwen3-VL-235B-A22B from 63.7 to 12.8 — consistent with MW's longer-horizon, cross-application, backend-driven task design (Section 3; 27.8 vs. 14.3 average steps; 62.2% vs. 9.5% multi-app). The single highest leaderboard SR on MW (82.1, Qwen-UI-Agent) and the leaderboard's Claude-Opus-4.7 entry (56.4) both exceed our Claude Code + Opus-4.7 CLI configuration (51.9); these leaderboard rows are reported under MW's native screenshot-based protocol and a different harness, so the comparison is across rather than within paradigm.

On-device vs. backend-assisted tasks on MobileWorld. MobileWorld's networked apps keep part of their task state on backend servers, which four of the twelve MobileWorld tools reach (`http`, `pg`, `backend-exec`, `service-status`; Table 6). These tools are benchmark infrastructure rather than part of the phone's own CLI. To separate the two, we split the 101 CLI-solvable MobileWorld tasks by whether their oracle solution uses a backend tool: 45 are solvable through on-device ADB alone and 56 are backend-assisted (Table 10). Every off-the-shelf CLI agent scores *lower* on the backend-assisted split than on the on-device split, so the aggregate CLI results are not carried by backend access; MobileCLI is the only configuration that does not lose ground there (84.5 against 80.7). The backend-assisted tasks cost the GUI baselines as well, and on average by a wider margin — Claude e2e falls from 69.6 to 48.8 and GUI-Owl-1.5-32B from 48.9 to 29.2 — since state held on a server is reachable through an app front end only over longer trajectories. On the on-device split, where neither paradigm touches benchmark infrastructure, the ordering of Table 1 is preserved at a fixed Opus 4.7 backbone: MobileCLI leads at 80.7%, Claude e2e follows at 69.6%, and Claude Code trails at 65.2%. The CLI's MobileWorld advantage is therefore a property of the purpose-built harness rather than of backend access, and closing the backend-assisted gap remains the main headroom for off-the-shelf harnesses.

Table 10: **MobileWorld success rate by access surface.** The 101 CLI-solvable tasks are split by whether the oracle solution uses a backend tool (Table 6): 45 are solvable through on-device ADB alone and 56 are backend-assisted. SR (%), mean (std) over 3 seeds. Every row re-aggregates the per-task results behind Table 1.

Agent	**Model API**	**On-device** (n=45)	**Backend-assisted** (n=56)	**All CLI-solvable** (n=101)	**Full MW** (n=117)
GUI agents (reproduced)					
Claude e2e	Opus 4.7	69.6 (3.4)	48.8 (1.0)	58.1 (1.5)	57.0 (1.3)
GUI-Owl-1.5-32B	–	48.9 (5.9)	29.2 (3.7)	38.0 (4.5)	41.3 (4.4)
MAI-UI-8B	–	43.0 (7.1)	17.3 (5.5)	28.7 (6.0)	27.4 (4.5)
Qwen3-VL-32B	–	20.7 (4.6)	7.1 (1.8)	13.2 (1.1)	15.1 (2.0)
CLI agents					
Claude Code	Claude Opus 4.7	65.2 (3.4)	56.0 (1.0)	60.1 (1.1)	51.9 (1.0)
mini-swe-agent	Claude Sonnet 4.6	61.5 (1.3)	41.7 (6.3)	50.5 (3.0)	43.6 (2.6)
Terminus-2	GPT-5.3 Codex	45.2 (1.3)	39.3 (1.8)	41.9 (1.1)	36.2 (1.0)
Terminus-2	MiniMax M2.7	29.6 (1.3)	11.3 (3.7)	19.5 (1.5)	16.8 (1.3)
MobileCLI	Opus 4.7	80.7 (1.3)	84.5 (1.0)	82.8 (1.1)	71.5 (1.0)

Table 11: **Full leaderboard-extended version of Table 1.** Metrics and seeds as in Table 1. Leaderboard rows report the published SR from the snapshots described in Appendix C.1. ‡ SR-CLI and Avg Step computed by us from the authors' released trajectories; these entries postdate the snapshot. "–": metric unavailable; "—": method absent from that benchmark's leaderboard.

		AndroidWorld			MobileWorld		
Method	**Model API / Notes**	**SR↑**	**SR-CLI↑**	**Avg Step↓**	**SR↑**	**SR-CLI↑**	**Avg Step↓**
Leaderboard-reported — General-purpose LLMs (no specialized GUI training)							
Seed-1.8	ByteDance, end-to-end	70.7	–	–	52.1	–	–
Seed-2.0-Pro	ByteDance, end-to-end	—	–	–	63.2	–	–
SeedVL-1.5	ByteDance, end-to-end	62.1	–	–	—	–	–
Claude-Sonnet-4.5	Anthropic, end-to-end	56.0	–	–	47.8	–	–
Claude-Opus-4.6	Anthropic, end-to-end	—	–	–	44.5	–	–
Claude-Opus-4.7	Anthropic, end-to-end	—	–	–	56.4	–	–
Gemini-2.5-Pro	Google, end-to-end	69.7	–	–	—	–	–
Gemini-3-Pro	Google, end-to-end	—	–	–	51.3	–	–
Gemini-3.1-Pro-Preview	Google, end-to-end	—	–	–	58.1	–	–
GPT-5.5	OpenAI, end-to-end	—	–	–	62.4	–	–
OpenAI CUA o3	OpenAI, end-to-end	31.3	–	–	—	–	–
Kimi-K2.5	Moonshot, end-to-end	—	–	–	49.6	–	–
Kimi-K2.6	Moonshot, end-to-end	—	–	–	55.6	–	–
Kimi-K3‡	Moonshot, end-to-end	—	–	–	74.4	74.3	25.4
Qwen3.5-397B-A17B	Alibaba, end-to-end	—	–	–	42.7	–	–
Qwen3-VL-8B-Instruct	Alibaba, end-to-end	47.6	–	–	9.4	–	–
Qwen3-VL-32B-Instruct	Alibaba, end-to-end	57.3	–	–	11.9	–	–
Qwen3-VL-235B-A22B-Instruct	Alibaba, end-to-end	63.7	–	–	12.8	–	–
Leaderboard-reported — GUI specialists (mobile post-trained)							
Qwen-UI-Agent‡	Alibaba	—	–	–	82.1	82.2	25.1
MobileRL-9B	Zhipu	80.2	–	–	—	–	–
UI-Venus-1.5-30B-A3B	Ant Group	77.6	–	–	17.1	–	–
UI-Venus-72B	Ant Group	—	–	–	16.4	–	–
UI-Venus-7B	Ant Group	—	–	–	8.5	–	–
MAI-UI-8B	Alibaba	70.7	–	–	27.5	–	–
MAI-UI-32B	Alibaba	73.3	–	–	36.2	–	–
MAI-UI-235B-A22B	Alibaba	76.7	–	–	39.7	–	–
UI-TARS-72B-DPO	ByteDance	46.6	–	–	—	–	–
UI-TARS-1.5	ByteDance	64.2	–	–	—	–	–
UI-TARS-2	ByteDance	73.3	–	–	—	–	–
Doubao-1.5-UI-TARS	ByteDance	—	–	–	26.3	–	–
Step-GUI-8B	StepFun	67.7	–	–	16.1	–	–
GELab-Zero-4B	StepFun	63.9	–	–	16.1	–	–
GUI-Owl-7B	Alibaba	66.4	–	–	7.7	–	–
GUI-Owl-32B	Alibaba	—	–	–	8.5	–	–
GUI-Owl-1.5-2B-Instruct	Alibaba	67.9	–	–	32.2	–	–
GUI-Owl-1.5-4B-Instruct	Alibaba	69.8	–	–	—	–	–
GUI-Owl-1.5-8B-Instruct	Alibaba	69.0	–	–	38.2	–	–
GUI-Owl-1.5-8B-Thinking	Alibaba	71.6	–	–	—	–	–
GUI-Owl-1.5-32B-Instruct	Alibaba	69.8	–	–	43.9	–	–
GUI-Owl-1.5-32B-Thinking	Alibaba	69.8	–	–	—	–	–
Leaderboard-reported — Agentic frameworks (planner + grounder composition)							
GPT-5 + UI-Ins-7B	OpenAI + Ant Group	—	–	–	54.0	–	–
Gemini-3-Pro + UI-Ins-7B	Google + Ant Group	—	–	–	55.6	–	–
Claude-4.5-Sonnet + UI-Ins-7B	Anthropic + Ant Group	—	–	–	47.8	–	–
Reproduced GUI baselines (our infrastructure)							
Claude e2e	Claude Opus 4.7	75.0 (0.9)	79.3 (1.1)	21.4 (0.0)	57.0 (1.3)	58.1 (1.5)	28.6 (0.6)
GUI-Owl-1.5-32B	–	69.3 (0.5)	71.6 (1.5)	15.5 (2.5)	41.3 (4.4)	38.0 (4.5)	34.7 (1.0)
MAI-UI-8B	–	68.1 (0.6)	69.3 (0.3)	18.4 (1.2)	27.4 (4.5)	28.7 (6.0)	38.5 (2.0)
Qwen3-VL-32B	–	57.2 (3.0)	62.8 (3.9)	18.4 (0.7)	15.1 (2.0)	13.2 (1.1)	24.4 (0.1)
CLI agents (ours)							
Claude Code	Claude Opus 4.7	71.8 (1.8)	80.9 (2.0)	15.2 (0.6)	51.9 (1.0)	60.1 (1.1)	30.2 (0.6)
mini-swe-agent	Claude Sonnet 4.6	70.7 (2.3)	79.6 (2.6)	12.1 (0.3)	43.6 (2.6)	50.5 (3.0)	33.5 (0.4)
Terminus-2	GPT-5.3 Codex	63.2 (0.5)	71.2 (0.6)	7.1 (0.1)	36.2 (1.0)	41.9 (1.1)	24.5 (1.9)
Terminus-2	MiniMax M2.7	55.2 (3.8)	62.1 (4.2)	14.5 (0.5)	16.8 (1.3)	19.5 (1.5)	32.9 (0.8)
MobileCLI	Claude Opus 4.7	79.0 (0.5)	89.0 (0.6)	7.4 (0.3)	71.5 (1.0)	82.8 (1.1)	18.3 (0.3)
Oracle (CLI ceiling)	–	88.8	100.0	3.7	86.3	100.0	5.6

Table 12: **Matched-backbone comparison on AndroidWorld. SR** is over all 116 tasks, **SR-CLI** over the 103 CLI-solvable tasks, mean (std) over 3 seeds. Claude Code runs `bash`-only. MobileCLI was tuned on MobileWorld with Opus 4.7 and is evaluated here with the backbone named in each block.

I/O interface	Agent	SR↑	SR-CLI↑	Steps↓
	Claude Opus 4.7			
GUI	end-to-end	75.0 (0.9)	79.3 (1.1)	21.4 (0.0)
CLI	Claude Code	71.8 (0.5)	80.9 (0.6)	13.0 (0.3)
CLI	MobileCLI	79.0 (0.5)	89.0 (0.6)	7.4 (0.3)
	Qwen3-VL-32B			
GUI	end-to-end	57.2 (3.0)	62.8 (3.9)	18.4 (0.7)
CLI	MobileCLI	34.8 (1.3)	39.2 (1.5)	17.0 (0.5)

C.2 Matched-model pairs

The paradigms move together across capability tiers. The strongest CLI results in Table 1 rely on frontier proprietary coding agents, as Section 6 states. The finding does not hinge on that single system. mini-swe-agent, a minimal open-source scaffold, reaches 70.7% with Sonnet 4.6, matching the best reproduced GUI baseline GUI-Owl-1.5-32B (69.3%) at better step efficiency, and the weakest CLI configuration (Terminus-2 with MiniMax M2.7, 55.2%) tracks the weakest reproduced GUI baseline (Qwen3-VL-32B, 57.2%). The two paradigms move together across capability tiers rather than separating at the frontier.

The unit of comparison. Each paradigm is instantiated as a model-and-harness configuration and evaluated as a system under fixed tasks, verifiers, and metrics. The main tables deliberately do not fix a shared model. The strongest GUI baselines are specialists post-trained on mobile GUI datasets and environments, whereas CLI agents rely on off-the-shelf coding capability, and no available model is trained for both regimes; fixing one model would relocate the confound to this post-training mismatch rather than remove it.

What a matched pair can and cannot show. While no single pairing removes the confound, a matched-model pair *bounds* the interface effect at fixed model capability. We therefore evaluate Claude Opus 4.7 and Qwen3-VL-32B under both paradigms on the same tasks, verifiers, and metrics, matching the model within each pair; Table 12 reports the result.

C.3 Tool design

Appendix A.2 specifies the CLI tool suite — four wrappers on AndroidWorld, twelve on MobileWorld. Every experiment after Table 1 runs the `bash`-only configuration; this subsection isolates what the wrappers buy across three harnesses and four model APIs (Table 13), and covers the design question behind those numbers: which failure class each wrapper is meant to absorb, and what the agent does differently once it is absorbed.

Tool gains are conditional on model strength. Table 13 compares bash-only and `+tools` modes across three CLI harnesses (Claude Code, mini-swe-agent, Terminus-2) and four model APIs on AndroidWorld. In bash-only mode, GPT-5.3 Codex is the weakest model API (48.9–49.7% SR), well below Claude Sonnet 4.6 (58.9–69.5%). Consistent with recent findings that tool gains are conditional on model strength (Lou et al., 2026), only GPT-5.3 Codex benefits substantially from the four CLI tools, with success rate rising by $+11.8$ points on mini-swe-agent and $+14.3$ points on Terminus-2; other configurations change by at most 1.4 points. One mechanism behind these effects is shell quoting: the four CLI tools handle escaping that bash invocations frequently mishandle. The two paired Codex trajectories below illustrate this: `+tools` converts a bash-only failure into a success (the SR gain) and eliminates retry steps on a single-quote edit (the step reduction).

Table 13: **AndroidWorld tool ablation across 3 harnesses and 4 model APIs.** `bash-only` vs. `+tools` (the four wrappers of Appendix A.2); success rate (%) over all 116 tasks, mean (std) over 3 seeds. Δ: change from `bash-only` to `+tools` in points; green improves, red regresses.

Harness	Model API	bash-only SR↑	+tools SR↑	Δ
Claude Code	Opus 4.7	71.8 (0.5)	71.8 (1.8)	0.0
	Sonnet 4.6	67.5 (2.2)	66.1 (4.3)	↓1.4
mini-swe-agent	Sonnet 4.6	69.5 (3.0)	70.7 (2.3)	↑1.2
	GPT-5.3 Codex	49.7 (0.5)	61.5 (3.0)	↑11.8
	MiniMax M2.7	57.2 (3.3)	58.3 (2.2)	↑1.1
Terminus-2	Sonnet 4.6	58.9 (4.1)	59.8 (3.3)	↑0.9
	GPT-5.3 Codex	48.9 (3.0)	63.2 (0.5)	↑14.3
	MiniMax M2.7	55.2 (3.0)	55.2 (3.8)	0.0

PAIRED TRAJECTORY EXAMPLES

We provide two paired Codex trajectories that pin all variables except the toolset (same harness, same model, same seed) and explain the two mechanisms behind Table 13. Table 14 (Task 082) is the SR mechanism: `bash-only` loses the trajectory to nested-quote and heredoc errors that prevent it from reading the schema, and ultimately writes `NULL` into fields the verifier expects to be `''`; `+tools` sidesteps the same quoting failures with two `sql` calls, observes the existing-row format, and inserts correctly. Table 15 (Task 052) is the step-efficiency mechanism: both runs reach reward 1, but `bash-only` burns ten steps recovering from a single apostrophe in the note body, while `+tools` writes the file in one `write-file` call (the wrapper base64-encodes the body internally). Both cases reproduce on a second harness; full traces are in the released artifact.

C.4 GENERALIZATION ACROSS OS AND APP VERSIONS

This subsection documents the cross-version study summarized in Section 5.2 and reported in Table 2. It covers how we built the sibling Android 17 environment, the shared evaluation protocol, the significance tests behind the reported changes, and the model-free oracle replay.

Environment construction. We build a sibling AndroidWorld image that upgrades the operating system from Android 13 (API 33) to Android 17 (API 37) and 17 of 20 applications, holding task instructions, task instances, verifier logic, system prompts, seeds, and the 50-step budget fixed. Among AndroidWorld's 20 applications, 3 third-party apps are already at the newest published build, so we upgrade the remaining 12 third-party apps and the 5 first-party apps that ship with the system image (Table 16). We add schema shims that adapt task initialization and verification to the new schemas for 3 upgraded apps (Broccoli, OpenTracks, and VLC). The shims touch only the benchmark code, never the agents or their prompts. We then verify that 115 of the 116 tasks remain solvable and scorable on Android 17, where task `ContactsNewContactDraft` always fails because its AndroidWorld verifier depends on an Android 13 screen layout.

Protocol. Every row of Table 2 follows the protocol of Table 1 on both images: seeds 7, 30, and 1234, the same 116 task instances per seed, a 50-step budget, and the `bash`-only configuration for every CLI harness.

Significance tests. Table 17 reports, for each row, a Welch t-test comparing the three Android 13 runs with the three Android 17 runs, with p-values Holm-corrected over the eight rows, and the Welch 95% confidence interval of the change. The drops of GUI-Owl, MAI-UI, and Claude e2e are significant, and no other change survives Holm correction.

Model-free oracle replay. For each interface we build a hand-verified solution on Android 13 and replay it unchanged on Android 17 with no model in the loop (seed 30, one run, the same task instances and 50-step budget). The CLI oracle is a sequence of `adb shell` commands for each of the 103 CLI-solvable tasks. The accessibility-tree oracle is one route per task written against the

Table 14: **Case 1 — Task 082 (Broccoli recipes), Codex.** Full `bash-only` (17-step) and `+tools` (11-step) trajectories; same harness, same seed. The SR shift (0 → 1) traces to repeated nested-quote and heredoc errors in steps 10–16 of `bash-only` that prevent it from reading the existing-row format, leading it to insert `NULL` where the verifier expects `''` (step 17). `+tools` reads the schema and existing-row format in two `sql` calls (steps 5–6) and inserts correctly (step 10).

`bash-only`

17 steps + finish, 221,689 input tokens, reward 0

#	Action → Result
1	`pm list packages | grep -Ei 'markor|broccoli'; find ... -iname recipes.txt` → both packages + `/sdcard/Documents/Markor/recipes.txt`
2	`cat recipes.txt; run-as broccoli ...` → recipes read; `run-as` blocked (not debuggable)
3	probe external dir + `cmd package resolve-content-provider` (4 guesses) → ext dir empty; `Unknown command`
4	list ext files + `awk -F'|' 'NR==1 || $4=="4 hrs"'` → empty; 3 four-hr rows found
5	probe 7 candidate `content://com.flauschcode.broccoli*` URIs → 6 `Could not find provider`, 1 `No result found`
6	`find` Broccoli-named files → only `recipes.txt`
7	`logcat -c; force-stop; monkey; logcat -d` → discovers DB at `/data/.../broccoli/databases/broccoli` from `FileUtils: chmod` log
8	`cmd package query-activities -a SEND -t text/plain` → matches `CreateAndEditRecipeActivity`
9	`am start ... SEND text/plain ... -es TEXT 'Title|desc|...'` → "supplied URL is malformed" (SEND handler scrapes URLs)
10	enumerate 7×7 action/MIME combos via nested `"$out"` → `/system/bin/sh: no closing quote`
11	retry intent enumeration with cleaner quoting → only SEND text/plain matches
12	direct `sqlite3 broccoli '.tables'` → tables listed
13	`adb shell "sqlite3 broccoli \".schema recipes; .schema categories...\""` → `Usage: .schema ?--indent? ?LIKE-PATTERN?` (nested-quoted dot-cmd)
14	heredoc `cat <<'SQL' | sqlite3 broccoli` → schema returned; sample queries fail with `no such column: id`
15	heredoc with `json_each(readfile(...))` + nested `substr/instr` parser → `Error: near "(": syntax error`; `json_each` not built
16	`awk -F'|' '... printf "INSERT...VALUES(...NULL...NULL...);" ...' | sqlite3` → `/system/bin/sh: no closing quote`
17	heredoc `.import recipes.txt _import_recipes; INSERT INTO recipes SELECT title, NULL, ..., NULL, ..., 0 FROM _import_recipes WHERE preparationTime='4 hrs'` → 3 rows inserted with `imageName=NULL, source=NULL`
18	`finish "Inserted 3 recipes ..."` → **Reward 0** — `imageName/source` should be `''`, not `NULL`

`+tools`

11 steps + finish, 92,826 input tokens, reward 1 (−58%)

#	Action → Result
1	`pm list packages | grep -Ei 'markor|broccoli'; find ... -name recipes.txt` → both packages + recipes.txt path
2	`cat recipes.txt; run-as broccoli ...; dumpsys package | grep provider` → recipes read; `run-as` blocked
3	probe ext dir + `find /sdcard *broccoli*` + recipe-file candidates → only `recipes.txt` matches
4	`ls /data/.../broccoli/databases + find ... -name '*.db'` → **`broccoli` DB world-readable**
5	`sql broccoli "SELECT type,name,sql FROM sqlite_master ..."` → full schema in one call; FTS triggers identified
6	`sql broccoli "SELECT recipeId, title, imageName, ..., source, ... FROM recipes LIMIT 10"` → **observes `imageName=''`, `source=''`** in every existing row
7	`sql broccoli "SELECT preparationTime, COUNT(*) FROM recipes GROUP BY preparationTime"` → 0 existing 4-hr rows; safe to insert
8	`python3 --version 2>/dev/null || echo NO_PYTHON` → `NO_PYTHON`
9	`awk -F'|' 'NR==1 || $4=="4 hrs" {print NR"|"$0}' recipes.txt` → exactly 3 source rows (Margherita, Grilled Cheese, Avocado Toast)
10	`sql broccoli "BEGIN; INSERT INTO recipes (..., imageName, ..., source, ...) VALUES ('...', '', ..., '', ...); ... COMMIT;"` → **3 rows inserted with `imageName=''`, `source=''`**
11	`am force-stop com.flauschcode.broccoli` → —
12	`finish "Added the 3 recipes ..."` → **Reward 1**

screen's accessibility tree, and the screen-coordinate oracle is that route realized on pixels; both cover all 116 tasks. Every oracle passes all of its tasks on Android 13. The replay re-issues the stored commands or actions on the Android 17 device without adaptation. An answer to an information-retrieval task is credited only if it comes from the Android 17 device: the CLI oracle parses its answer from the Android 17 command output, and a GUI route's typed answer counts only if the route reaches a screen that shows it.

Table 15: **Case 2 — Task 052 (Markor note with apostrophe), Codex.** Full `bash-only` (17-step) and `+tools` (7-step) trajectories. Both runs reach reward 1; the step difference comes from `bash-only` burning steps 7–13 escaping a single apostrophe in the note body (combined with an `&&`-in-`adb shell` parsing interaction). `+tools`'s `write-file` base64-encodes content internally, so neither the apostrophe nor any other content character is ever exposed to shell parsing.

`bash-only`

17 steps + finish, 445,700 input tokens, reward 1

#	Action → Result
1	`pm list packages | grep -i markor` → `net.gsantner.markor`
2	`ls /sdcard/Documents/; ls /sdcard/Markor/; ls /sdcard/ | head -30` → finds `Markor` under `/sdcard/Documents/`
3	`ls -la /sdcard/Markor/` → no such directory
4	`ls -la /sdcard/Documents/` → confirms `Markor/` subfolder
5	`ls -la /sdcard/Documents/Markor/` → empty
6	`run-as net.gsantner.markor cat shared_prefs/*.xml | grep dir|folder|path` → (no output)
7	`adb shell 'echo "Lunch ... Cafe L'"'"'amour." > /sdcard/.../eHwd_helpful_jacket.md && ls -la ... && cat ...'` → `no closing quote; adb: ls requires an argument; adb: unknown command cat` (`&&` parsed as separate `adb` subcommands)
8	`adb shell 'cat > /sdcard/.../eHwd_helpful_jacket.md'` → (no output, stdin not connected)
9	`ls -la && cat` chain → same quoting error
10	`adb shell ls -la /sdcard/Documents/Markor/` → file exists but **0 bytes**
11	`adb shell 'cat > /sdcard/.../...'` → still empty
12	`adb shell ls -la ...` → 0 bytes
13	`adb shell 'echo "Lunch meeting ... Cafe L'"'"'amour." > /sdcard/.../...'` (no `&&` chain) → written
14	`adb shell ls -la ...` → **47 bytes**
15	`adb shell cat ...` → `Lunch meeting with Sarah at 1 PM Cafe L'amour.`
16	`am broadcast MEDIA_SCANNER_SCAN_FILE` → `Broadcast completed: result=0`
17	`am force-stop net.gsantner.markor` → —
18	`finish "Created note ..."` → **Reward 1**

`+tools`

7 steps + finish, 242,191 input tokens, reward 1 (−46%)

#	Action → Result
1	`pm list packages | grep -i markor` → `net.gsantner.markor`
2	`run-as markor ls shared_prefs/; ls /sdcard/Documents/; ls /sdcard/Markor/; ls /sdcard/` (one batched probe) → `run-as` blocked; `Markor` under `/sdcard/Documents/`
3	`ls -la /sdcard/Documents/Markor/` → empty
4	`find-files /sdcard "*.md"` → (no output — confirms no stray notes)
5	`write-file /sdcard/.../eHwd_helpful_jacket.md` content: `Lunch meeting with Sarah at 1 PM Cafe L'amour.` → written (wrapper base64-encodes body internally; **apostrophe never enters the shell**)
6	`read-file /sdcard/.../eHwd_helpful_jacket.md` → `Lunch meeting ... Cafe L'amour.` (apostrophe preserved)
7	`am broadcast MEDIA_SCANNER_SCAN_FILE; am force-stop net.gsantner.markor` → `Broadcast completed: result=0`
8	`finish "Created ... with content: ..."` → **Reward 1**

The CLI oracle retains 96 of its 103 solutions (82.8% of the 116 tasks, against 88.8% on Android 13). Six of the seven failures come from an OpenTracks database schema change that breaks the stored SQL queries. The accessibility-tree oracle retains 67 of 116 routes (57.8%) and the screen-coordinate oracle 49 of 116 (42.2%); nearly all of their failures are UI drift, concentrated in Simple Calendar Pro (14 of the failed routes in each arm), Markor (8 and 13), Settings (7 and 9), and OpenTracks (6 and 6): controls renamed or moved by new rows, month grids that start on a different day, and screens that no longer expose their contents to the tree. The coordinate replay fails wherever a shifted layout sends a fixed tap to a neighbouring element, which is why it retains less than the tree-based replay.

D Trajectory-Level Error Analysis — Details

This appendix details the full taxonomy (§D.1), the construction pipeline together with its reliability and judge validation (§D.2).

Table 16: **Android 13 versus Android 17 environments.** Version codes as reported by the package manager; release names in parentheses. Held fixed across both: screen 1080×2400 @ 420 dpi; AndroidWorld and `android_env` packages; verifier logic; task instances; seeds; prompts; 50-step budget.

	Android 13	**Android 17**
Platform		
System image (`google_apis`, x86_64)	API 33	API 37.0 rev. 6
Emulator / platform-tools	35.3.11 / 34.0.4	37.2.6 / 37.0.1
Launcher, Settings, system UI	Android 13	Android 17 (redesigned)
Third-party apps, upgraded (12)		
Markor	146	163 (2.16.1)
Simple Calendar Pro	238	240 (6.23.1)
Tasks.org	130605	150904 (15.9.1)
Simple Draw Pro	79	82 (6.9.6)
Audio Recorder	926	949 (2.4.0)
Pro Expense	11	12 (1.0.0-beta06)
Broccoli	1020600	1040600 (1.4.6)
OsmAnd	4.6.13	5403 (5.4.3)
OpenTracks	5705	6741 (4.28.2)
VLC	13050408	13070108 (3.7.1)
Joplin	2097740	2097810 (3.6.22)
Retro Music	10603	10703 (7.0.3)
Third-party apps already at the newest build (3)		
Simple Gallery Pro	396	unchanged
Simple SMS Messenger	85	unchanged
Clipper	1.2.1	unchanged
First-party apps, upgraded with the image (5)		
Chrome, Camera, Clock, Contacts, Files	Android 13 builds	Android 17 builds

Table 17: **Tests behind Table 2.** SR (%), mean (std) over 3 runs per side; Δ in points as in Table 2. Holm-corrected Welch p over the eight rows, and the Welch 95% confidence interval of the change (uncorrected).

Agent	**I/O**	**Android 13**	**Android 17**	**Δ**	**Holm p**	**95% CI**
MobileCLI + Opus 4.7	CLI	79.0 (0.5)	80.5 (0.5)	$+1.5$	0.11	$[+0.4, +2.6]$
Claude Code + Opus 4.7	CLI	71.8 (0.5)	71.0 (4.1)	-0.8	0.81	$[-10.8, +9.2]$
mini-swe-agent + MiniMax M2.7	CLI	57.2 (3.3)	59.2 (1.0)	$+2.0$	0.81	$[-5.4, +9.4]$
Terminus-2 + GPT-5.3 Codex	CLI	48.9 (3.0)	51.7 (0.9)	$+2.8$	0.73	$[-4.0, +9.6]$
Claude e2e + Opus 4.7	GUI	75.0 (0.9)	71.0 (1.0)	-4.0	0.042	$[-6.2, -1.8]$
GUI-Owl-1.5-32B	GUI	69.3 (0.5)	40.8 (1.0)	-28.5	< 0.001	$[-30.6, -26.4]$
MAI-UI-8B	GUI	68.1 (0.6)	35.1 (1.3)	-33.0	< 0.001	$[-35.7, -30.3]$
Qwen3-VL-32B	GUI	57.2 (3.0)	50.6 (1.8)	-6.6	0.17	$[-12.7, -0.5]$

D.1 Full taxonomy

This section defines the shared classification scale used by the trajectory-level error analysis in Section 5.3 across both comparison axes: (i) GUI vs. CLI paradigms (Figure 2(a)) and (ii) `bash-only` vs. `+tools` for GPT-5.3 Codex (Figure 2(b)). How the scale was instantiated on our trajectory pool — the Docent clustering setup, the Opus 4.7 cluster→leaf mapping, and the two-annotator rubric calibration — is documented separately in Appendix D.2.

Two-level paradigm-agnostic taxonomy. The paradigm-agnostic backbone is a two-level adaptation of MAST (Cemri et al., 2026), single-agent subset, following Terminal-Bench's mobile adaptation (Merrill et al., 2026). *Level 1* is three top-level error classes — *Execution*, *Coherence*, *Verification* — that partition the agent's cognitive loop. *Level 2*, underneath each class, is a small set of **named failure modes** (the MAST *leaves*); the nine MAST leaves we inherit retain their original semantics but are re-stated in Android-specific terms (Table 18). All nine leaves are defined for

both paradigms with identical operational meanings; only their prevalence differs. Using the same Level-1/Level-2 scale for both paradigms is what lets us write a single classification statement about any trajectory regardless of action interface.

Paradigm-specific clusters underneath the taxonomy. Below the two-level taxonomy we further annotate each trajectory with a Docent (Meng et al., 2025) cluster that captures the concrete mobile mechanism behind the failure. These clusters are paradigm-specific because the action interfaces themselves are paradigm-specific: a CLI *wrong write surface* has no GUI analogue, and a GUI *wrong-row tap* has no CLI analogue. The bottom-up clustering pass surfaced 20 CLI clusters and 18 GUI clusters (Tables 19 and 20); every classified trajectory is assigned to exactly one cluster, and each cluster's primary MAST leaf is fixed by the mapping step in Appendix D.2. Clusters therefore play the role of a paradigm-specific *third* layer beneath the paradigm-agnostic taxonomy: they refine, but do not extend, the MAST leaves.

Top-level classes and leaves. Table 18 lists the three classes, the nine MAST-derived leaves grouped under them, the operational definition of each leaf, and the primary-label prevalence we observe. Five leaves are active in at least one paradigm; four are uniformly zero in our setting (rationale at the end of this section). We retain the four inactive leaves in the published taxonomy so that future longer-horizon or multi-agent extensions can populate them without redefining the tree.

Paradigm-specific clusters — CLI. Table 19 enumerates the 20 CLI clusters grouped under each active leaf (Disobey Specification, Reasoning-Action Mismatch, Premature Termination, Weak Verification). The identifiers are the canonical cluster names used throughout our released artefact. The third column gives the dominant Android mechanism behind the failure; the **Share** column reports each cluster's share of the classified CLI pool.

Paradigm-specific clusters — GUI. Table 20 enumerates the 18 GUI clusters grouped under each active leaf (Disobey Specification, Step Repetition, Premature Termination); none of the four Coherence or remaining Verification leaves are activated, consistent with the per-leaf zeros in Table 18. The GUI mass is more concentrated than the CLI mass: a single Step-Repetition cluster (`identical_action_loop_until_step_budget_exhausted`) absorbs more than a quarter of all GUI failures, and three clusters above 10% jointly account for over half of the pool.

Departures from MAST. We follow Terminal-Bench in dropping MAST's inter-agent coordination subcategories (*Specification Issues*, *Inter-Agent Misalignment*) because all our agents are single-agent. Within the remaining tree, the four uniformly-zero leaves in Table 18 have a single-agent rationale: *Unaware of Termination* does not fire because both paradigms expose an explicit `finish`/`submit` action with hard semantics; *Context Loss* and *Task Derailment* do not fire because our tasks are short-horizon (≤ 50 steps) and the goal text is re-displayed every step in all three harnesses; *No or Incorrect Verification* is empirically subsumed by Weak Verification (verifying on the wrong surface) in our setting, because the cases where the verification *logic* is malformed turn out to coincide with cases where the verification *surface* is also wrong.

D.2 Construction

This appendix documents how the taxonomy in §D.1 was instantiated on our trajectory pool and how its reliability was validated.

Pipeline. The labelling pipeline runs in three stages:

- **Per-trajectory failure summary.** Each rollout is distilled into a structured, rubric-shaped paragraph by Claude Opus 4.7, recording the agent's surface, value, and verification choices.
- **Bottom-up clustering (Docent (Meng et al., 2025)).** Docent agglomeratively clusters the per-trajectory summaries within each paradigm separately, yielding the paradigm-specific clusters listed in Tables 19 and 20.
- **Cluster → leaf mapping.** Claude Opus 4.7 reads each cluster's exemplars and assigns it a single MAST leaf, using a rubric that was *calibrated against two human annotators* on a 20-trajectory calibration subset before being released to the rest of the pool.

Table 18: **Two-level paradigm-agnostic failure taxonomy: three top-level MAST classes and the nine MAST-derived leaves underneath.** The **CLI %** and **GUI %** columns give the primary-label share of failed trajectories assigned to each leaf (active-leaf shares sum to 100% within each paradigm). The **#CLI** and **#GUI** columns count the paradigm-specific clusters under each leaf (Tables 19 and 20). Bolded leaves are active in at least one paradigm; **DS** is the cross-paradigm dominant mode. Construction details are in Appendix D.2.

Class	Leaf	Operational definition	CLI %	#CLI	GUI %	#GUI
Execution	**Disobey Specification (DS)**	Action is well-formed at the interface but does not satisfy the verifier contract — wrong surface, wrong value, or wrong artifact for the same goal.	**76.7**	16	**40.4**	11
	Step Repetition (SR)	Same action repeated on the same observation with no progress; loop continues until the step budget exhausts.	0.0	0	**35.9**	4
	Unaware of Termination	Agent does not realize the task is done and keeps issuing actions past the success state.	0.0	0	0.0	0
Coherence	**Reasoning-Action Mismatch (RAM)**	The action does not implement the predicate the reasoning describes; the trajectory commits to a divergent action because it runs cleanly.	**14.0**	2	0.0	0
	Context Loss	Earlier observation or instruction is no longer reflected in the agent's state; the agent acts on stale context.	0.0	0	0.0	0
	Task Derailment	Agent veers to a related but unrequested goal in the middle of the trajectory.	0.0	0	0.0	0
Verification	**Premature Termination (PT)**	Agent calls `finish`/`submit` without a post-action read of the device state; the closing handshake is decoupled from what actually happened.	**5.7**	1	**23.7**	3
	No or Incorrect Verification	A verification command is issued but its result is logically mis-interpreted (e.g., empty result read as success).	0.0	0	0.0	0
	Weak Verification (WV)	Verification reads the same surface that was just written, while the grader reads a different surface; the agent's confirmation is non-informative for the actual contract.	**3.6**	1	0.0	0

Docent setup. We use Docent (Meng et al., 2025) (`v0.4.2`) with its default setup. The CLI and GUI failure pools are clustered separately so that paradigm-specific mechanisms surface as distinct clusters. GUI trajectories are pre-processed by rendering each rollout as an action+observation transcript: action JSONs are kept verbatim and the corresponding accessibility-tree screenshots are replaced by a one-sentence Claude-Opus 4.7 caption of the rendered screen, keeping the input length comparable to a CLI rollout. Docent surfaces 20 CLI clusters and 18 GUI clusters; singletons are discarded and the residual "noise" bucket (cluster -1) accounts for 4.1% of CLI failures and 2.4% of GUI failures, which we exclude from the per-cluster prevalence numbers.

Cluster→leaf prompt (verbatim). The Opus 4.7 judge receives the prompt in Figure 4 for every cluster. `{cluster_id}` is the canonical Docent name, `{exemplars}` is a random sample of eight trajectory summaries drawn from the cluster, and the two rubric files are passed verbatim in the system prompt.

```
You are mapping a Docent failure-mode cluster onto our two-level mobile failure taxonomy:
3 top-level classes (Execution, Coherence, Verification) and 9 named leaves underneath.
The taxonomy is single-agent MAST adapted to Android per Terminal-Bench.

INPUTS
  cluster_id     : canonical Docent cluster name
  exemplars      : 8 per-trajectory failure summaries from this cluster
  rubric         : rubric_v2.md (one-sentence operational definition of each of the
                   9 leaves; Android-specific framings)
  tie_breakers   : rubric_v2_clarifications.md (16 numbered rules that resolve the
                   most common 2-leaf ambiguities; each rule has a pattern,
                   a "maps to" leaf, and an example)

TASK
  1. Read the 8 exemplars. State the dominant Android failure mechanism in one
     sentence, in your own words (no taxonomy jargon).
  2. Walk the tree top-down. Which top-level class (Execution, Coherence, or
     Verification) best describes the failure? Justify in one sentence.
  3. Within the chosen class, pick the single best-fitting leaf.
  4. If two leaves are equally plausible, consult tie_breakers and cite the rule
     number that decides (e.g. "TB7: timezone misinterpretation -> Disobey
     Specification, wrong output protocol").
  5. Optionally record one secondary leaf, only if at least 3 of the 8 exemplars
     also support it. Never list more than one secondary.

OUTPUT  (strict JSON, no prose outside the object)
  {
    "cluster_id":     "...",
    "mechanism":      "one sentence in plain English",
    "primary_class":  "Execution | Coherence | Verification",
    "primary_leaf":   "...",
    "tie_breaker":    "TB# or null",
    "secondary_leaf": "... or null",
    "rationale":      "2-4 sentences grounded in specific exemplars"
  }
```

Figure 4: **Verbatim Opus 4.7 prompt for the cluster→leaf mapping step (Appendix D.2).** `rubric_v2.md` and `rubric_v2_clarifications.md` are passed in the system prompt. The judge runs in JSON mode with temperature 0, reasoning effort `high`, and a maximum of 4,096 output tokens.

Rubric structure. The rubric (`rubric_v2.md`) is a flat, 9-row file: one row per MAST leaf, each row giving (i) a one-sentence operational definition, (ii) an Android-specific framing in the spirit of Section §D.1, and (iii) 2–3 positive exemplars and 1–2 negative exemplars drawn from the calibration subset (see below). The assignment procedure is the top-down tree walk in the prompt above: a cluster is first placed in one of three top-level classes (Execution/Coherence/Verification), then in the best-fitting leaf within that class. The companion file `rubric_v2_clarifications.md` contains 16 numbered tie-breakers covering ambiguities that surfaced during calibration — e.g. same-surface verification (DS vs. WV), timezone misinterpretation (DS, wrong value), Q&A from the wrong provider (PT vs. WV), GUI long-press idiom (DS, wrong API level), single-slot clipboard during multi-source merge (DS, wrong protocol). Each tie-breaker is a single paragraph stating the pattern, the leaf it maps to, and a worked example, and is referenced by number from the judge prompt.

Inter-annotator agreement. On the 20-trajectory calibration subset, two human annotators (one author and one external researcher with 5+ years of Android-development experience, blinded to the cluster identities) independently assigned each trajectory a top-level class. They agreed on $18/20$ trajectories (raw agreement $P_o = 0.90$); the two disagreements were both at the Coherence/Execution boundary. Cohen's $\kappa = 0.85$ (95% CI $[0.62, 1.00]$, 1,000-sample bootstrap) — the gap from P_o to κ reflects the chance-agreement correction at three roughly balanced classes ($P_e \approx 1/3$).

Judge validation against human labels. On a disjoint, held-out validation set of $M = 60$ trajectories (20 per top-level class, stratified by paradigm and harness), the Opus 4.7 judge agrees with the consensus human top-level-class label on $56/60$ trajectories ($P_o = 0.933$); Cohen's $\kappa = 0.90$ (95% CI $[0.80, 0.98]$, 1,000-sample bootstrap), at or above the inter-human $\kappa = 0.85$ on the calibration subset, so we treat the judge as a substitute for a third human annotator.

Cluster→leaf mapping table (released artefact). The full 38-row mapping — every Docent cluster, its assigned primary leaf, the tie-breaker invoked (if any), the secondary leaf (if any), and a flag for whether the Opus 4.7 judge agreed with both human annotators — is too large to reproduce inline; it is released as `cluster_to_leaf_mapping.csv` in the analysis bundle. Across all 38 clusters, the judge agreed with the consensus human label on 36 (94.7%).

E LIMITATIONS OF THE CLI PARADIGM

All experiments in this paper run on Android emulators and containerized backend services provisioned with synthetic benchmark data. No real user data was read, transmitted, or stored at any point. The risks described below are therefore properties of the paradigm at deployment, not of the experiments we report.

We flag three risks specific to deploying CLI mobile agents outside the benchmark setting: two concern the read surface and one the action surface. Appendix A.3 details what each paradigm accesses and at what privilege level.

Unmediated access to user-private storage. On a real device the CLI surface reads user-private storage directly through `adb shell` and `content` queries, without the permission prompts, SAF intents, or provider-authority checks that mediate ordinary app access; any deployed CLI agent is therefore a privileged on-device process that sees the user's data in clear form by construction. The user has no corresponding way to inspect or revoke this access through Android's permission UI, since the grant is made once at the ADB level rather than per resource.

Off-device transmission. The agents we evaluate are driven by frontier model APIs hosted in the cloud, which means that whatever the on-device CLI reads — including the user-private data above and any task context that quotes it — is sent off-device to the model provider on every step; a careless deployment leaks user information to a third party.

A privileged action surface. The shell-level grant that widens the read surface also widens the action surface. Commands available to the agent by default include operations that irreversibly destroy user data, of which `pm clear` is the most consequential: it wipes an application's entire private storage. This is not hypothetical. Our failure taxonomy records `pm clear` as a failure cluster under Disobey Specification (Appendix D.1), issued as a blanket recovery action after repeated failures rather than as a targeted step. In the benchmark setting the cost is a failed episode; on a user's device it would be data loss.

Mitigations. These risks are mitigable, but not by the agent alone. Any deployment beyond the benchmark setting requires an allowlisted command surface, with destructive and privacy-sensitive commands gated behind explicit user consent. Mechanisms for granting shell-level access without a permanently attached host already exist in consumer software: delegated-ADB services such as Shizuku[1] expose the same interfaces to an authorized client app after a one-time activation, and an OEM-integrated assistant would hold comparable privileges by construction. These narrow the provisioning barrier but not the exposure: an agent reaching user data through a delegated grant reads it just as directly, and a cloud-hosted model still receives it. Removing that exposure requires moving inference on device, which is the small-model direction we identify as future work (Section 6).

[1] `https://github.com/RikkaApps/Shizuku`

Table 19: **Paradigm-specific CLI clusters grouped under their assigned MAST leaf.** The **Share** column reports each cluster's primary-label share of the classified CLI pool; within-leaf shares sum to the leaf totals in Table 18, and cross-leaf shares sum to 100%. Cluster→leaf assignments, the rubric and tie-breakers, and the two-annotator agreement statistics are reported in Appendix D.2.

Leaf	Docent cluster (canonical name)	Dominant Android mechanism	Share
DS	wrote_to_wrong_database_surface	Persisted state to a SQLite file or content provider that the target app does not consume (e.g., wrote SMS rows to AOSP `mmssms.db` when the verifier reads Simple SMS's own `conversations.db`).	20.7%
	ocr_or_vision_tool_hunt_without_writing_target	Burned the entire step budget probing for OCR / vision tools (`tesseract`, `strings`/`xxd` over image bytes, MediaStore description columns) and never executed the target `INSERT` or `write-file`.	8.9%
	task_required_gui_only_no_cli_pathway	Task ground truth is gated by an on-screen tap or rendered DOM state with no shell write surface (Chrome canvas verifiers, Camera shutter, AudioRecorder UI). Agent computed answers offline via `awk`/`bc` or fabricated artifacts but the verifier reads the live foreground UI.	8.2%
	fabricated_artifact_instead_of_invoking_app_pipeline	Hand-crafted the artifact (base64 JPEG, `screenrecord` MP4, GPX with invented coords, raw contacts2.db `INSERT`) instead of driving the responsible app to produce real provenance bytes/rows.	8.2%
	permission_or_role_blocked_clipboard_or_sms	Looped on shell calls to clipboard / SMS surfaces blocked by Android 13's permission gate, often misreading a `SecurityException` parcel as "clipboard empty" rather than pivoting to a foreground app or the default-SMS role.	6.3%
	time_or_timezone_misinterpretation	Encoded ambiguous time literals as the wrong period (PM vs. AM, UTC vs. device-local), or used overlap (`start<end AND end>start`) instead of containment in time-window queries.	4.4%
	recurrence_or_filter_predicate_omission	SQL or content-query predicate dropped a required column the schema exposed: `repeat_interval` on recurring events, `dueDate` vs. `completed` on org.tasks, `hideUntil`/`importance`/`parent` on subtasks.	4.0%
	byte_exact_file_content_mismatch	Markor merge/edit/paste produced almost-right bytes but introduced `\n\n` separators where a single `\n` was required, dropped a trailing newline, or substituted file extension; `cat`-based self-checks could not catch the byte-level disagreement.	3.0%
	apk_static_analysis_loop_without_db_write	Locked into 30–50 turns of `strings`/`unzip -p`/`xxd`/`dexdump` against `base.apk` trying to recover an enum-or-ordinal, and never executed the required `INSERT`.	3.0%
	truncated_input_treated_as_complete	Read a source file or dump where the harness cut output mid-row, then synthesized invented continuations rather than paginating with `sed`/`tail`/`wc -l` or re-issuing with a projection.	2.8%
	wrong_notebook_root_for_markor	Inferred Markor's notebook directory from the presence of seeded `.md` files instead of reading `pref_key__notebook_directory` from shared_prefs, writing notes one directory level too deep or too shallow.	2.3%
	screenrecord_or_screencap_substitute_for_camera	Substituted `screenrecord` MP4 or `screencap -p` PNG (sometimes renamed `.jpg`) for the camera shutter; verifier checked `MediaStore.owner_package_name` and rejected the screen-capture provenance.	2.3%
	pm_clear_or_destructive_blanket_action	Used `pm clear`, `rm -rf` of an entire directory, or `DELETE FROM sqlite_master` where the verifier expected a single-row delete; wiped sibling data that was meant to remain intact.	0.8%
	shell_quoting_or_harness_parse_blocked_action	Command died in the wrapper due to nested-quote / apostrophe issues (`L'amour` inside an inner-shell `INSERT`), multi-JSON-per-turn parse errors, or `max_tokens` truncation; zero commands reached the device.	0.6%
	chrome_firstrun_or_file_scheme_blockade	Burned steps fighting Chrome's `FirstRunActivity` via shared_prefs edits and `am start ChromeTabbedActivity` hacks instead of tapping the FRE accept button through Files-app→Chrome.	0.6%
	harness_verb_or_command_prefix_violation	Emitted bare Android verbs (`settings`, `cmd`, `svc`, `am`) without the `adb shell` prefix the wrapper requires; the parser rejected every command before it reached the device.	0.4%
RAM	treated_broadcast_or_intent_dispatch_as_state_change	Fired `am broadcast`/`am start` with an invented or AOSP-only action string and accepted `result=0` or a `dumpsys` top-activity match as proof of the state change, without inspecting persisted shared_prefs / SQLite / UI.	7.6%
	inverted_or_guessed_enum_mapping	Wrote a guessed integer for an unknown enum (`importance` where 0 means *High*; `repeat_rule` weekday bitmask; Pro Expense `category`; DeskClock `timer_state_0`) after reasoning admitted the mapping was uncertain.	6.3%
PT	answer_question_through_wrong_provider	For app-specific Q&A (Simple Calendar Pro, OpenTracks, org.tasks), queried the AOSP system provider, accepted `No result found.` as final, and answered `None`; never tried the app-private SQLite under `/data/data/<pkg>/databases/`.	5.7%
WV	self_confirming_select_loop	After any write (`sqlite3 INSERT`, `content insert`, `settings put`), verified success by re-reading the exact same surface with the inverse verb, never round-tripping through the consumer's actual read path (`content query`, app relaunch, MediaStore re-scan).	3.6%

Table 20: **Paradigm-specific GUI clusters grouped under their assigned MAST leaf.** The **Share** column reports each cluster's primary-label share of the classified GUI pool; within-leaf shares sum to the leaf totals in Table 18, and cross-leaf shares sum to 100%. The GUI mass concentrates in Step Repetition and Premature Termination ($35.9 + 23.7 = 59.6\%$), whereas the CLI mass concentrates in Disobey Specification (76.7%): the same paradigm-agnostic two-level tree separates the two action interfaces cleanly without redefining any leaf. Construction details are in Appendix D.2.

Leaf	Docent cluster (canonical name)	Dominant Android mechanism	Share
DS	skipped_source_image_then_fabricated_destination_entries	Launched the viewer app for a named source media file, switched apps within 1–2 steps with no full-screen render observed, then typed invented destination values (Pro Expense rows, Broccoli recipe lines, Markor transcriptions).	11.8%
	missing_two_step_record_or_count_observation	Collapsed an observe-between-actions pattern (start-record/stop-record; tap-then-count) into back-to-back taps at the same coordinate with no intervening observation, so no artifact or correct aggregate is produced.	4.9%
	stuck_on_search_or_filter_dialog_with_wrong_field_type	Typed human-readable date strings or category phrases into a title-only search field (Tasks, Broccoli, OpenTracks), got empty results, then answered `0/""` or looped on the same wrong query.	4.5%
	deletion_or_move_targeted_wrong_row_or_first_match_only	Long-pressed a visually similar but wrong filename (`2023_02_13_shy_king_copy.md` instead of `shy_king_copy.md`) or deleted only the first visible row of a multi-row category.	4.5%
	wrong_menu_path_for_markor_rename	Cycled through Markor's editor overflow menu (*File settings* → *Format*) instead of long-pressing the file-list row, so the filename never changes.	4.1%
	missing_long_press_gesture_for_selection_or_marker	Used `click` or `scroll` where the Android idiom requires `long_press` (Markor delete, OsmAnd marker, VLC multi-select, text-selection handles) and looped on the wrong gesture indefinitely.	2.4%
	single_tap_on_html_canvas_instead_of_swipe_stroke	On the `task.html` drawing subtask in Chrome, issued a single `click` at one canvas coordinate per colour instead of a `swipe`/drag, so no stroke was registered and the canvas was submitted blank.	2.0%
	wrong_clock_face_digit_in_time_picker	On Simple Calendar Pro's Material 24-hour TimePicker, tapped a digit on the wrong ring leaving the default 16:00 unchanged, or set only the end time without touching start.	2.0%
	answer_string_emitted_as_unknown_action_or_omitted_entirely	Had the correct comma-separated titles in reasoning but emitted them under `action_type:"unknown"/"status"/"wait"` instead of `"answer"`; verifier reports a missing answer payload.	1.6%
	single_clipboard_slot_overwritten_during_multi_source_merge	Chained *Copy* across all source files before any *Paste*, so each new Copy overwrote Android's single-slot clipboard and final Pastes delivered only the last file's content.	1.6%
	sent_message_in_wrong_conversation_thread	Read the source SMS in the sender's thread, then composed and sent the reply inside the same thread rather than backing out to start a new conversation with the named recipient.	0.8%
SR	identical_action_loop_until_step_budget_exhausted	Fired the exact same action (click at one coordinate, scroll in one direction, `wait`, `open_app`) for 30–50 consecutive steps with byte-identical reasoning, never branching when the screen failed to advance.	27.3%
	fixed_coordinate_delete_macro_on_reflowing_list	Recorded a 4-tap delete sequence (row long-press → overflow → Delete → Confirm) at hard-coded y-coordinates on Broccoli/Markor and replayed it 5–9 times without re-reading the list as rows shifted upward.	5.3%
	horizontal_chip_row_scroll_failure_in_pro_expense	Issued ≥ 10 consecutive `scroll(left)` (or alternating left/right) actions against Pro Expense's category chip strip, never tapping a visible chip or trying vertical scroll, never reaching SAVE.	2.0%
	calendar_chevron_oscillation_without_view_switch	Alternated forward and backward chevron taps in the Calendar day/month header for 30–44 steps, undoing its own navigation, never switching to week/agenda view or emitting an `answer`.	1.2%
PT	answered_immediately_after_open_app_without_reading_screen	For retrieval tasks, emitted an `answer` action one step after `open_app` with no scroll/click/wait in between, returning a guessed number, a fabricated comma-separated list, or `0`.	11.8%
	declared_complete_before_dialog_confirmed	Filled a dialog (folder name, brightness slider, time picker, delete confirmation) and emitted `status=complete` before tapping the final OK/confirm button or screenshotting the post-tap state.	9.0%
	filled_only_one_form_instance_for_multi_item_task	Task required N items (recipes, transactions, expenses) but the agent saved only one entry and declared complete, or treated a multi-recipe source file as a single copy-paste into one Title field.	2.9%